

**COMSATS University Islamabad, Lahore Campus**  
**Department of Computer Engineering**



# **CPE345 – Embedded Systems Workshop**

Lab Manual for Fall 2024 & Onwards

---

**Lab Resource Person**

Engr. Usman Rafiq

**Supervised By**

**Name:** \_\_\_\_\_ **Registration Number:** *CIIT/* \_\_\_\_\_ - \_\_\_\_\_ - \_\_\_\_\_ */LHR*

**Program:** \_\_\_\_\_ **Batch:** \_\_\_\_\_

**Semester** \_\_\_\_\_

## Revision History

Sr. No.	Update	Date	Performed by
1	Lab Manual Review	Sep-17	Engr. Mayyda Mukhtar
2	Lab Manual Preparation	Feb-18	Engr. Zainab Akhtar Engr. Noman Naeem
3	Lab Manual Review	Mar-18	Dr. Muhammad Naeem Shehzad
4	Lab Manual Format Changes	Sep-18	Engr. Muzamil Ahmad Zain
5	Lab Manual Review	Sep-19	Engr. Faisal Tariq
6	Lab Manual Update (removal of erroneous tables and programs)	Feb-20	Engr. Muhammad Usman Rafique Engr. Moazzam Ali Sahi
7	Rearrangement of Lab Experiments	Feb-21	Engr. Moazzam Ali Sahi Engr. M. Hassan Aslam
8	Lab Manual Modification	August-21	Engr. Nesruminallah
9	Rearrangement of Lab Experiments	Jan-22	Engr. Moazzam Ali Sahi Engr. M. Hassan Aslam
10	Updated with New Experiments	March-25	Engr. Usman Rafiq

## Preface

Embedded Systems Workshop (CPE345) is a lab-centric course which is intended for undergraduate students building embedded systems using microcontrollers, sensors and actuators. The purpose of this course is to present the design methodology of real embedded systems to train young engineers to understand the basic building blocks of microcontroller-based electronic systems enabling them to use these basic building blocks to design small embedded system design projects.

This course is intended to teach sixth semester undergraduate students of Computer Engineering about the components of an embedded system, design methodology, developing high-level language programs for microcontrollers, understanding the basic working principles of common sensors and actuators, and interfacing them microcontrollers.

The microcontroller used for the lab work is ESP32 which is a 32-bit microcontroller. The integrated development environment (IDE) used for developing and downloading executable code into the microcontroller is Espressif IDE. Starting from the introduction of ESP32 microcontroller, the lab work smoothly discusses the programming and interfacing methodologies that control the GPIOs, ADCs, DACs, SPI devices and I2C devices. Initially, student learns the code writing, compiling, debugging, and downloading into the ESP32 flash memory in a step-by-step manner. Once he is skilled on this exercise, the subsequent labs become more challenging that exploit features of Espressif IDE along with on-chip peripherals of ESP32. Several common devices such as pushbuttons, LEDs, matrix keypad, DC motor, servo motor, stepper motor, accelerometer, LCD, temperature sensor etc. are interfaced with ESP32 to set the course for designing advanced embedded systems.

By the of the workshop, students must submit an advanced level project that implements the key concepts and skills learned during this workshop as a Complex Engineering Problem (CEP).

## Books

### Textbook

1. ESP32 Technical Reference Manual Version 5.2

### Reference Books

1. Kolban's Book on ESP32 By Neil Kolban
2. Hand-outs provided by instructor

## Learning Outcomes

### Lab CLOs:

1. Assemble and configure the hardware components of an embedded system, utilizing microcontrollers and peripheral devices, while employing appropriate C programming techniques. (P4-PLO5)
2. Design microcontroller-based embedded systems by integrating hardware and software components to meet specified requirements. (C5 - PLO 4)
3. Prepare a comprehensive project report and demonstrate a functional embedded system, explaining its design, implementation, and testing process. Justifies design choices, follows standard reporting practices, and invites discussion through a structured presentation. (A3-PLO10)

## CLOs - PLOs Mapping

CLO \ PLO									Cognitive Domain	Affective Domain	Psychomotor Domain
	PLO1	PLO2	PLO4	PLO5	PLO6	PLO7	PLO10				
CLO1				X							P4
CLO2			X						C5		
CLO3							X			A3	

## Lab CLOs - Lab Experiment Mapping

CLO \ Lab												
	Lab 1	Lab 2	Lab 3	Lab 4	Lab 5	Lab 6	Lab 7	Lab 8	Lab 9	Lab 10	Lab 11	Lab 12
CLO 1	P2	P2	P2	P3	P4	P4	P4	P4	P4	P4	P4	P4

## Grading Policy

### Complex Engineering Problem Lab [CEP]

Lab Assignments: i. Lab Assignment 1 Marks (Lab marks from Labs 1-3) ii. Lab Assignment 2 Marks (Lab marks from Labs 4-6) iii. Lab Assignment 3 Marks (Lab marks from Labs 7-9) iv. Lab Assignment 4 Marks (Lab marks from Labs 10-12)	25%
Lab Mid Term = $0.5 * (\text{Lab Mid Term exam}) + 0.5 * (\text{average of lab evaluation of Lab 1-6})$	25%
Lab Terminal = $0.5 * (\text{Complex Engineering Problem}) + 0.375 * (\text{average of lab evaluation of Lab 7-12}) + 0.125 * (\text{average of lab evaluation of Lab 1-6})$	50%
<b>Total (lab)</b>	<b>100%</b>

## List of Equipment

- ESP32 Microcontroller Board
- Pushbuttons
- LEDs
- 4x3 keypad
- 20x4 I2C-controlled LCD
- Potentiometer
- Stepper motor
- Servo motor

## Software Resources

- Espressif IDE
- Microsoft Windows 10 Operating System

## Lab Instructions

- This lab activity comprises three parts: Pre-lab, Lab Tasks, Lab Report, Conclusion and Viva Voce session.
- The students should perform and demonstrate each lab task separately for stepwise evaluation.
- Only those tasks that are completed during the allocated lab time will be credited to the students.
- Students are, however, encouraged to practice on their own in spare time for enhancing their skills.

## Disciplinary and Safety Instructions

1. Turn off or put your cell phones on silent.
2. Avoid shouting and similar nonserious behavior.
3. Place your bags in the allocated rack.
4. Do not leave the lab without seeking permission from the lab instructor.
5. Log in with your username and password for your use only. Never share your username and password with others.
6. Edibles and applying cosmetics are not allowed in the computer lab or anywhere near a computer.
7. Respect the equipment. Do not remove or disconnect parts, cables, or labels.
8. Do not reconfigure the cabling/equipment without prior permission.
9. Internet use is limited to teacher assigned activities or classwork only.
10. Personal internet use for chat rooms, instant messaging (IM), or email is strictly prohibited. (This is against our Acceptable Use Policy).
11. Do not download or install any program, game, or music. (This is against our Acceptable Use Policy.)
12. No internet/intranet gaming activities allowed.
13. Do not personalize the computer settings. (This includes desktop, screen saver, etc.)
14. If, by mistake, you get to an inappropriate internet site, turn off your monitor immediately and raise your hand.
15. Do not run such programs that continue to run after you log out.
16. Upon completion of your work, log out your workstation.
17. Do not leave your workstation unattended. Do not leave processes in the background without prior approval from the Systems Manager. Do not lock your workstation for more than 20 minutes.



## Table of Contents

Revision History .....	i
Preface .....	ii
Books .....	iii
Learning Outcomes .....	iii
CLOs – PLOs Mapping .....	iii
Lab CLOs – Lab Experiment Mapping .....	iii
Grading Policy .....	iv
List of Equipment .....	iv
Software Resources .....	iv
Lab Instructions .....	iv
Disciplinary and Safety Instructions .....	v
LAB # 1.....	9
To Explain the Control of GPIOs and Interfacing External LEDs and Pushbuttons with ESP32 Microcontroller using C++ .....	9
Objectives .....	9
Pre-Lab Exercise .....	9
LAB # 2.....	21
To Explain the Working of UART of ESP32 and its Programming using C++ .....	21
Objectives .....	21
Pre-Lab Exercise .....	21
LAB # 3.....	28
To Explain the Working of ADC of ESP32 and its Programming using C++ .....	28
Objectives .....	28
Pre-Lab Exercise .....	28
LAB # 4.....	33
To Follow the I2C Protocol and Interfacing LCD with ESP32 using C++ .....	33
Objectives .....	33
Pre-Lab Exercise .....	33
LAB # 5.....	43
To Construct C++ Code for I2C Link to Interface RTC and LCD with ESP32 .....	43
Objectives .....	43



Pre-Lab Exercise .....	43
<b>LAB # 6.....</b>	<b>59</b>
<b>To Construct C++ Code for SPI Interface and Matrix Keypad .....</b>	<b>59</b>
Objectives .....	59
Pre-Lab Exercise .....	59
<b>LAB # 7.....</b>	<b>74</b>
<b>To Manipulate the External Interrupts and Timers on ESP32 using C++.....</b>	<b>74</b>
Objectives .....	74
Pre-Lab Exercise .....	74
<b>LAB # 8.....</b>	<b>82</b>
<b>To Construct C++ Code for on-chip Digital to Analog Converter of ESP32.....</b>	<b>82</b>
Objectives .....	82
Pre-Lab Exercise .....	82
<b>LAB # 9.....</b>	<b>87</b>
<b>To Construct C++ Code for Interfacing MicroSD Card with ESP32.....</b>	<b>87</b>
Objectives .....	87
Pre-Lab Exercise .....	87
<b>LAB # 10.....</b>	<b>94</b>
<b>To Construct C++ Code for Interfacing Sensors and GPS Receiver with ESP32.....</b>	<b>94</b>
Objectives .....	94
Pre-Lab Exercise .....	94
<b>LAB # 11.....</b>	<b>98</b>
<b>To Construct C++ Code for Interfacing Stepper Motor with ESP32.....</b>	<b>98</b>
Objectives .....	98
Pre-Lab Exercise .....	98
<b>LAB # 12.....</b>	<b>102</b>
<b>To Interface Radio Frequency (RF) Module with ESP32 using C Programming.....</b>	<b>102</b>
Objectives .....	100
Pre-Lab Exercise .....	100

## LAB # 1

### To Explain the Control of GPIOs and Interfacing External LEDs and Pushbuttons with ESP32 Microcontroller using C++

#### Objectives

- To understand the working with Espressif IDE package
- To create, build and run a C/C++ project in Espressif IDE and ESP32
- To understand C/C++ code for controlling general-purpose inputs/outputs (GPIOs) on ESP32
- To interface LEDs with ESP32
- To interface pushbuttons with ESP32

#### Pre-Lab Exercise

Read the details given below to comprehend the working with Espressif IDE and GPIOs of ESP32 microcontroller.


#### Working with Espressif IDE

Espressif IDE is a complete integrated development environment (IDE) that facilitates writing, editing, compiling, assembling and downloading the source code written for ESP32 microcontroller unit (MCU) in the MCUs flash memory. It allows the source code, written in C/C++ language for developing real-time embedded systems software, to debug and emulate before finalizing the software.

##### ➤ Creating a Project in Espressif IDE

A project is an abstract collection of files which are required to construct the executable binary file for MCU from the source code. A project provides a complete manipulation of the objectives and outcomes of an intended embedded system under development.

Creating Project in Espressif IDE (called E-IDE hereinafter) consists of certain steps that are to be carefully followed. Since this activity will be part of every lab work, students are expected to carefully observe the procedure.

**Step 1:** Launch the E-IDE program by double-clicking on the icon . This will open a window as shown in Figure 1.1.

**Step 2:** Create an empty folder in your working drive and give it an appropriate name such as Lab\_1 etc. This folder will be the main directory where all the project files are to be saved.

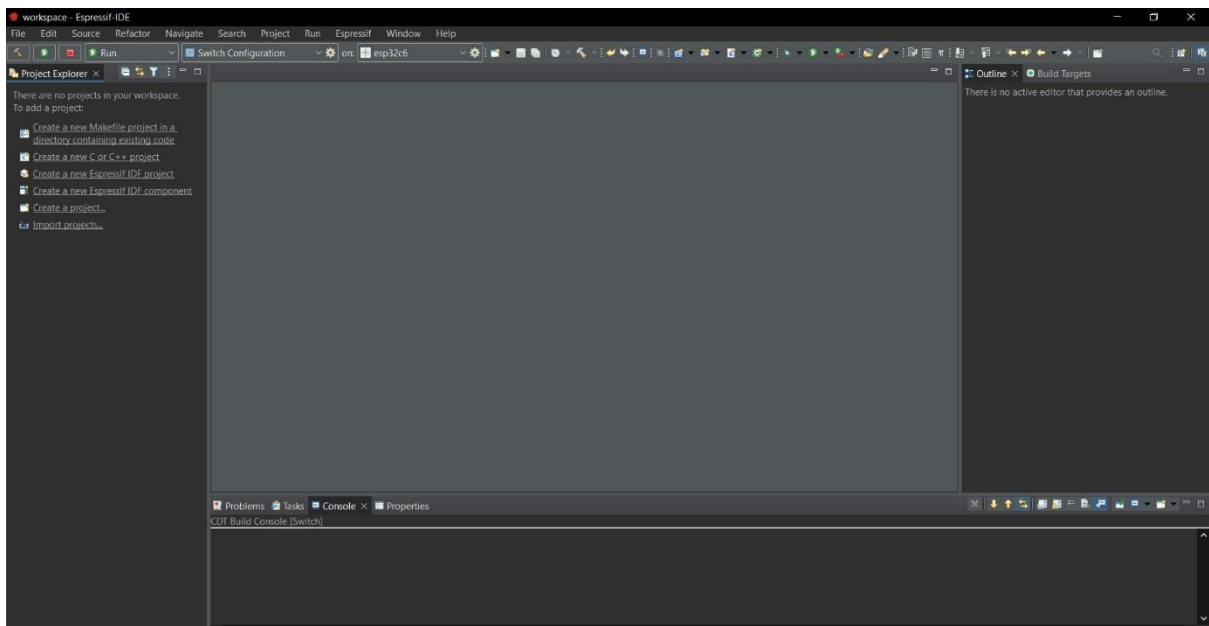


Figure 1.1

**Step 3:** From the File (top left corner of the main window (shown in Figure 1.1)), select 'New→Espressif IDF Project'. Following interface will open (shown in Figure 1.2).

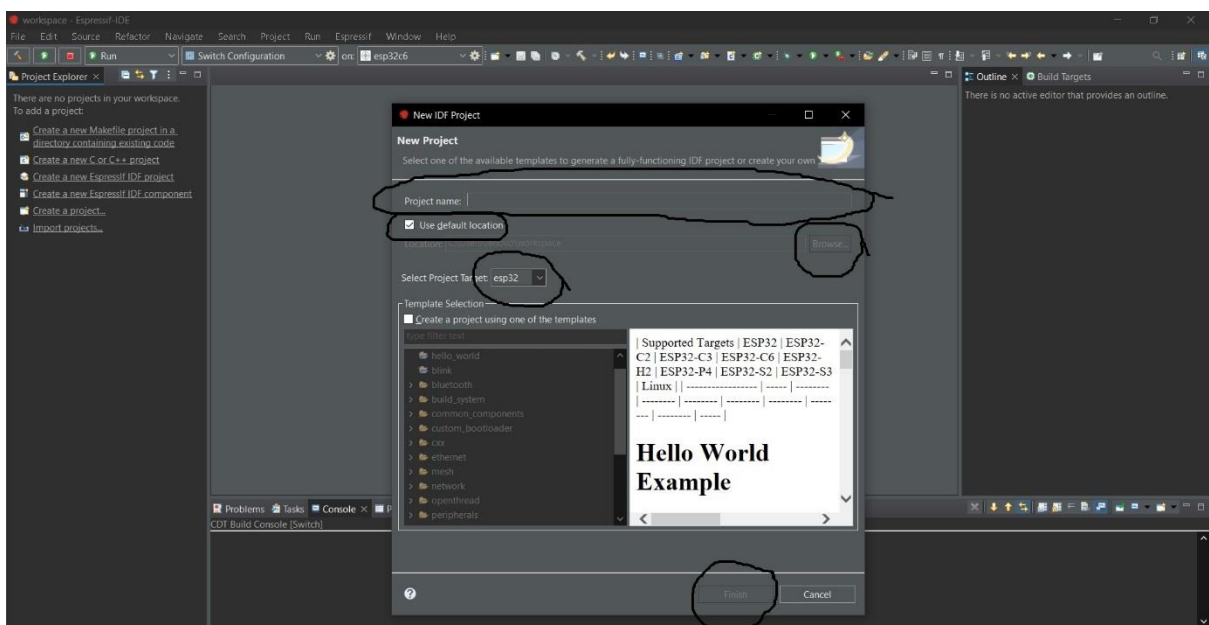


Figure 1.2

Give a name to the project by typing the name in 'Project name' bar, as shown in Figure 1.2. Uncheck 'Use default location' check box and give the path to your working directory by clicking on 'browse' button. From the 'Select Project Target' dropdown list, select 'esp32'. The project dialog box should appear as the one shown in Figure 1.3. If so, then click on 'Finish' at the bottom of this dialog box.

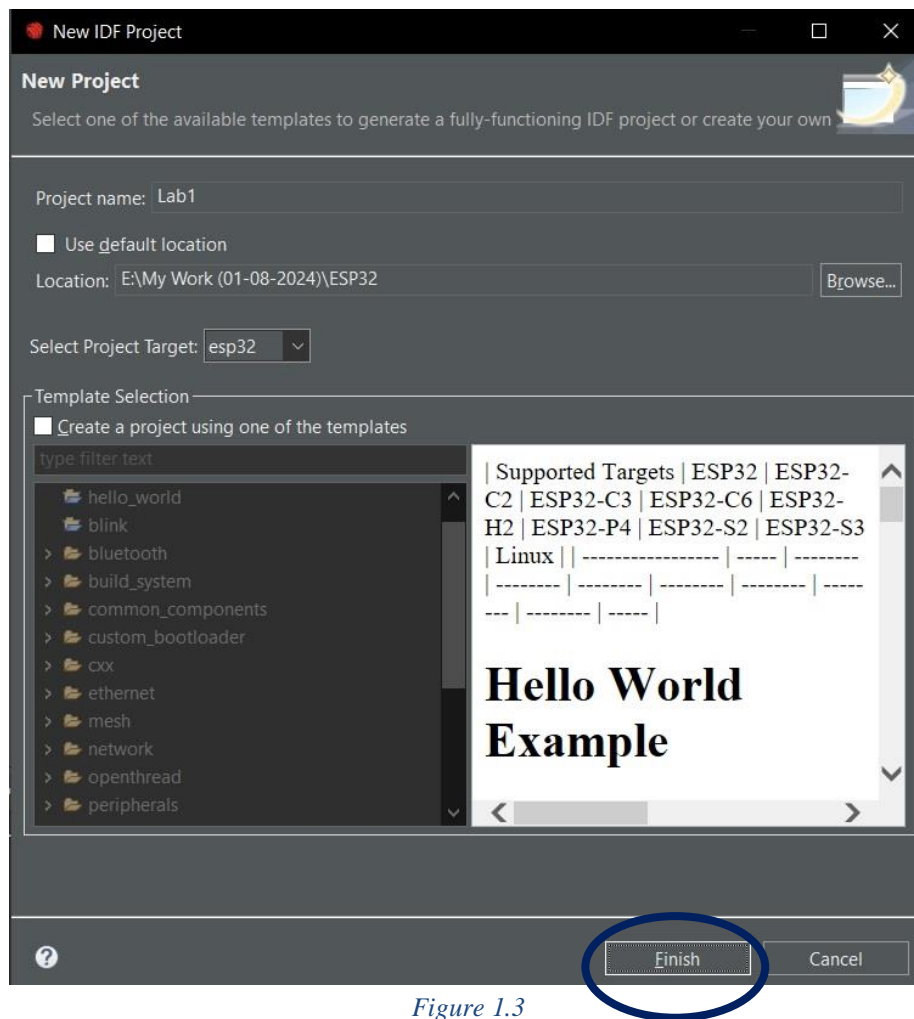


Figure 1.3

Check your working directory now. It should contain some files and a folder with name 'main'. Open this folder and there will be a '.c' file. ***Do not delete any of its content unless specified.***

**Step 4:** Refer to the box shown in top left of Figure 1.4. Click on the name of your project. In this figure, it is 'Lab1'. Then click on the 'main' and click on the 'main.c' file. This file will be opened in the editor, as shown in the Figure 1.4. The Project has been made, and C/C++ code is ready to be compiled.

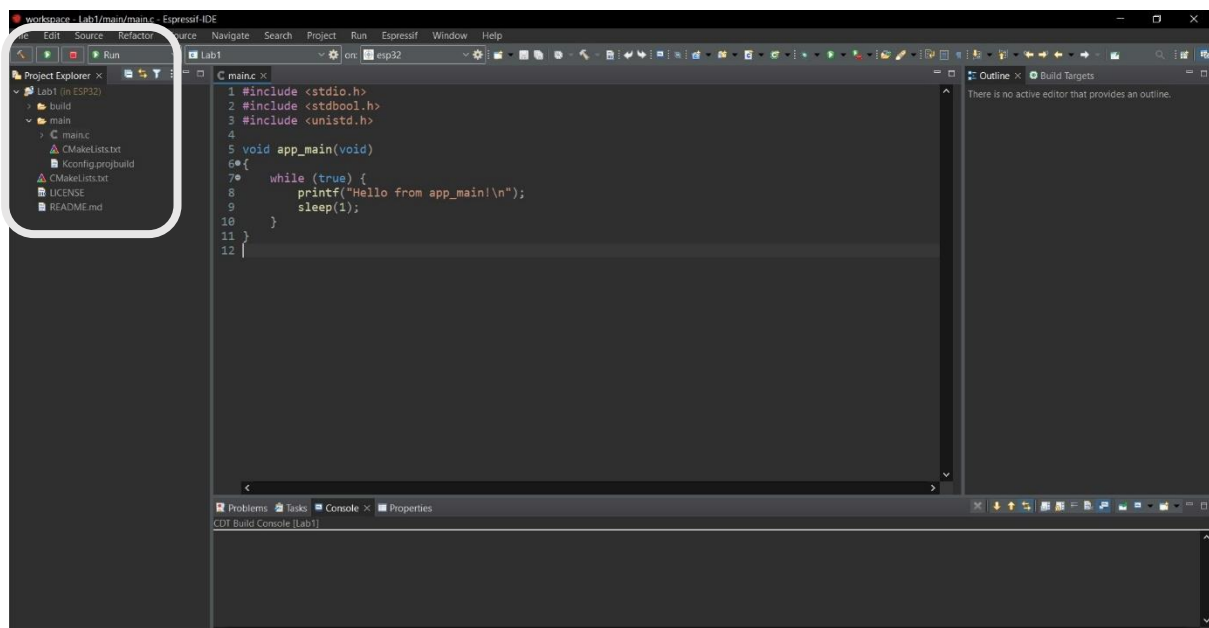


Figure 1.4

### ➤ Building the Project

Once the project is made and the source code is written, the project can be built. The process of ‘building a project’ is a sequence of several events that are controlled by the IDE. During the project building, source code is compiled, assembled and linked to generate the executable binary file that MCU runs. To understand the ‘project building’ process, it is assumed that the existing code, as shown in Figure 1.4, is our source code. However, this source code will be different for different projects.

For building the project, refer to the top left corner of window shown in Figure 1.5. Clicking on the icon will start ‘project build’ process. After the process starts, a sequence of messages appears at the lower end of the window, as shown in Figure 1.5. It takes around 5 to 6 minutes to build a new project. However, once the project is built, the rebuilding process finishes in 12 to 20 seconds. The ‘project build’ process can be halted anytime by pressing icon, located on the right of icon.

If there are no syntax errors in the source code, the ‘project build’ process will finish with the message as shown below in Figure 1.6. This message will appear at the bottom of the windows shown in Figure 1.5. If there are errors, remove them and rebuild the project by clicking on icon.

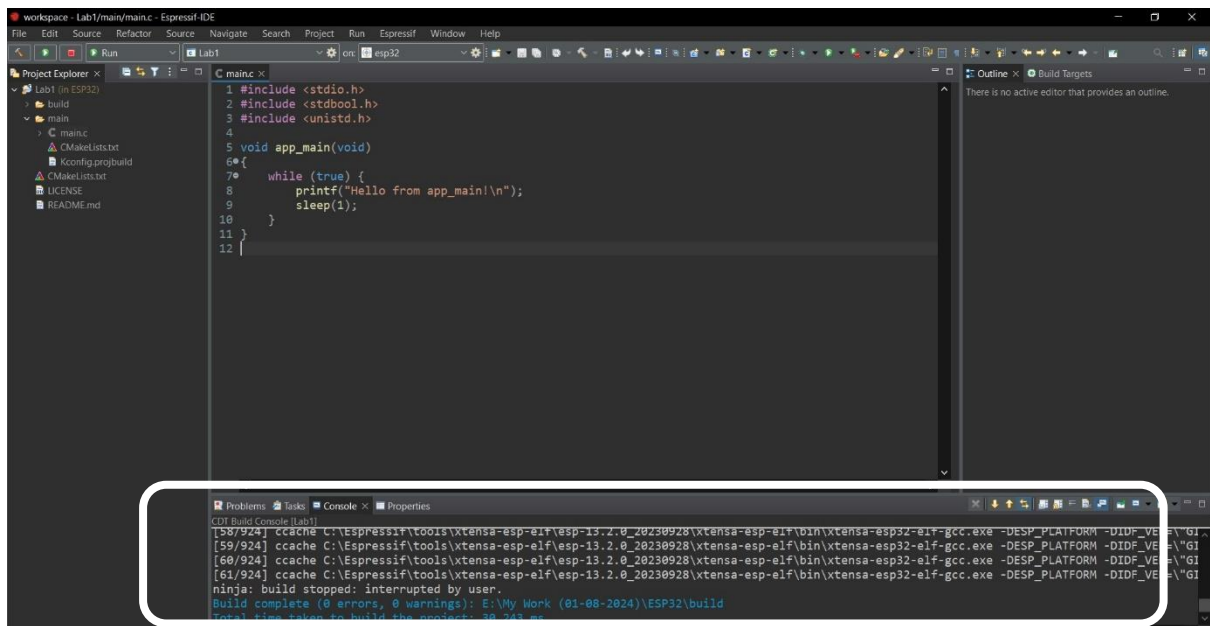


Figure 1.5

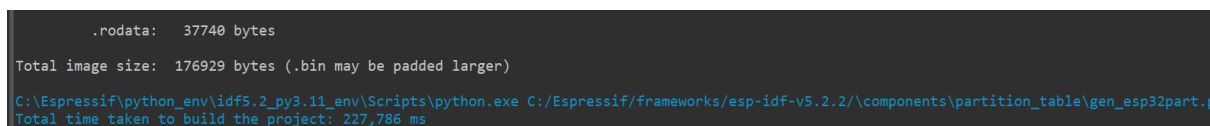



Figure 1.6

## ➤ Running the Project

Upon building the project successfully, the executable file can be downloaded in the flash memory of the target MCU. Now connect the ESP32 board with your computer using USB cable. The ESP32 board is shown in Figure 1.7.



Figure 1.7

Carefully plug in the proper end of USB cable with the USB connector provided on the board. Forcing the connector can damage it. An on-board red LED should be turned on. Refer to Figure 1.8 and click the  icon. This action will open a dialog box as shown in Figure 1.9.

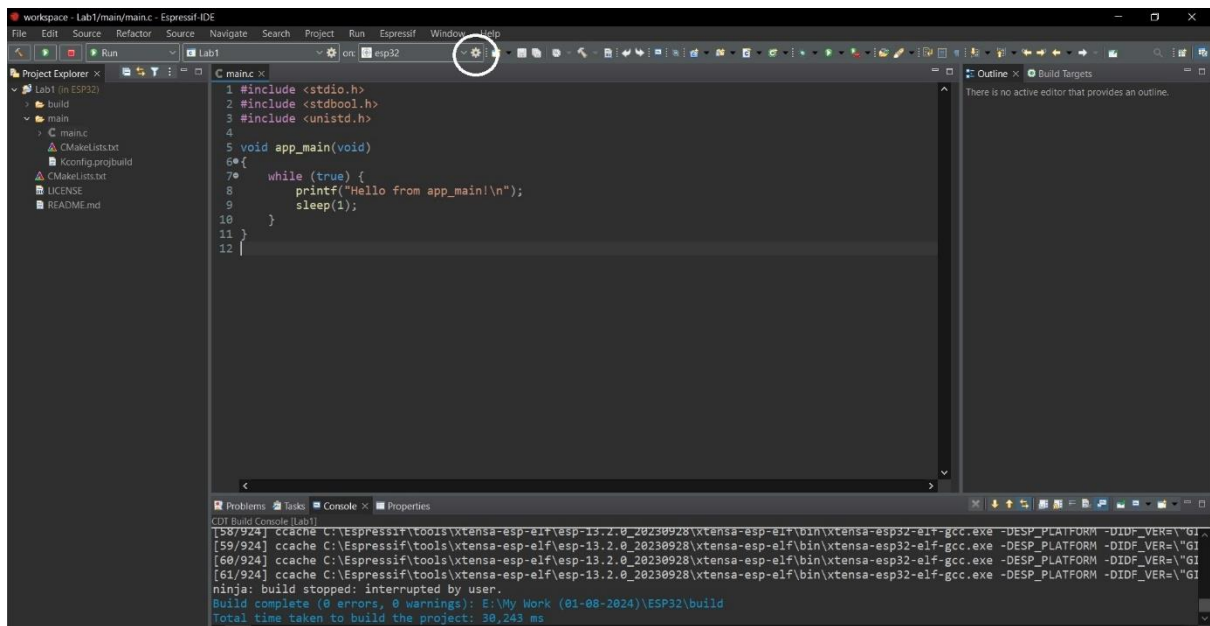


Figure 1.8

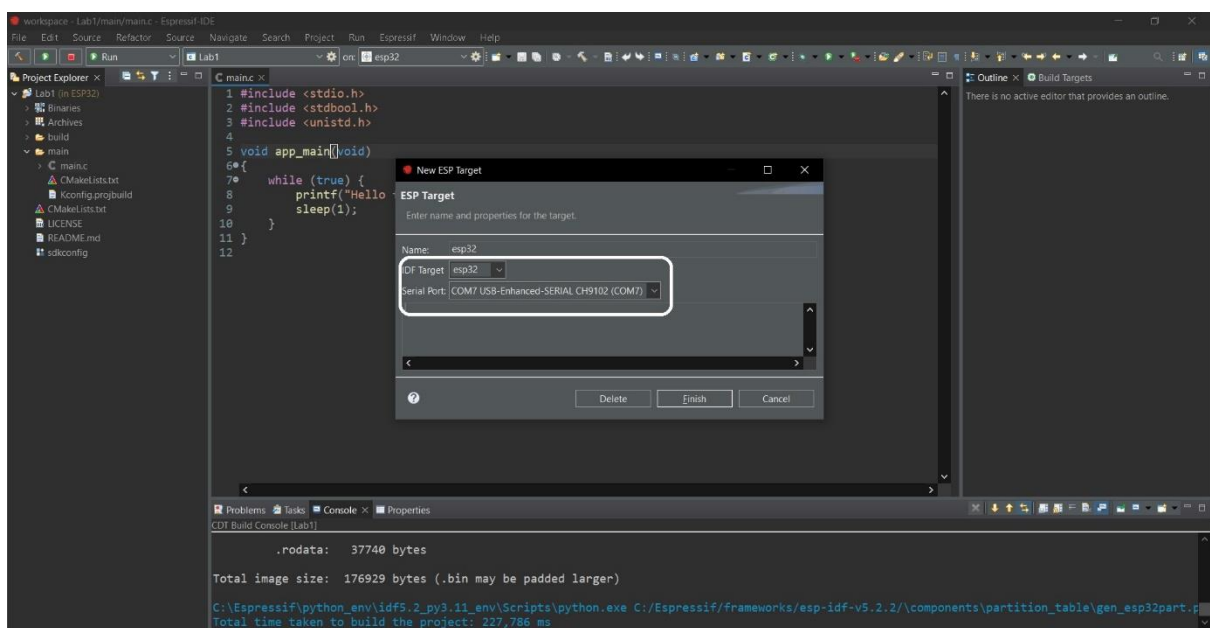




Figure 1.9

In this dialog box check the IDF Target is 'esp32' and the Serial Port, as shown in Figure 1.9. The COM port (serial port) that is shown against the Serial Port is the communication link between your host computer and the target MCU.

If they are shown, then click 'Finish' at the bottom of this dialog box. The dialog box will disappear now.



Now click on the icon  (located right of the icon ) to start ‘running’ the project. This will start a sequence of several steps that can take around 4 to 5 minutes for completion. Next time the project is ‘run’, it will take around 15 seconds for completion. If all is OK, then the executable file will be downloaded to the target MCU, and the ‘Done’ message will appear in the messages section. Program auto-starts running upon downloading.

### Pinouts of ESP32 DevKit V1 Board

ESP32 DevKit V1 is a small size circuit board that contains ESP32 MCU, two pushbuttons, two 15-pin male headers (one on either side), two LEDs and a wireless chip for radio communication. The pinout structure of this board is shown in Figure 1.10.

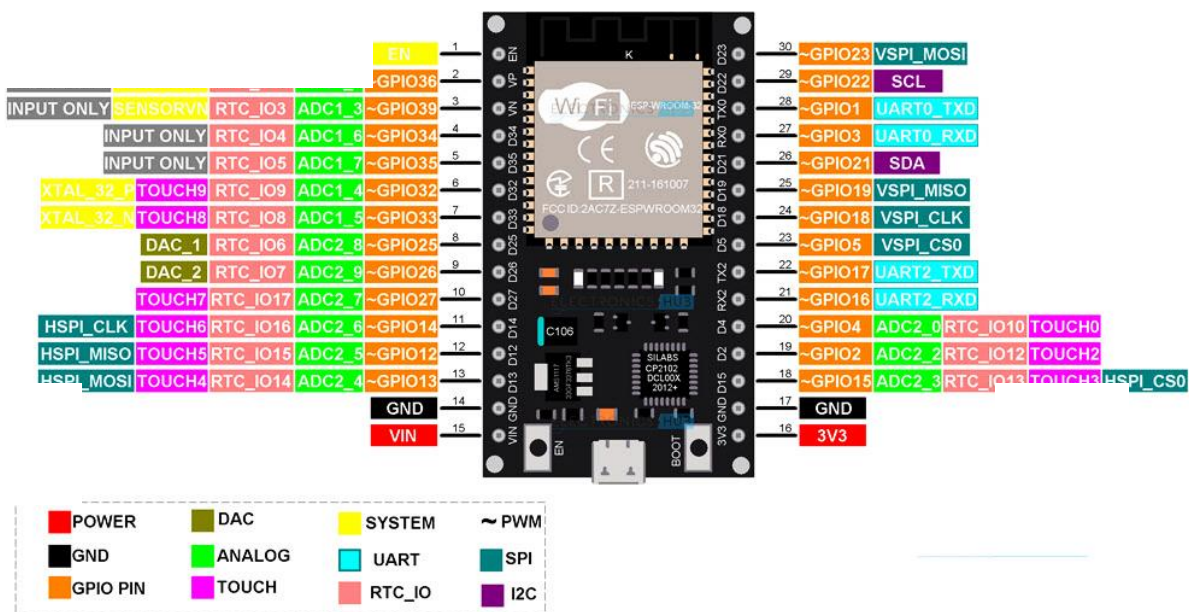


Figure 1.10

There are 30 pins on the circuit board, and each pin is accessible externally through male header. This male header can be inserted into a breadboard for connecting other components with ESP32 board. It can be inferred from Figure 1.10 that some pins have multiple functions. These functions are software configurable. A large group of pins are GPIOs. These are the digital input/output (IO) lines that can give or receive discrete voltage levels. A logic 1 is identified as 3.3V whereas logic 0 is identified as 0V.

### Program 1:

This program demonstrates the control of GPIOs, related libraries and testing the code by creating, building and downloading the executable file in ESP32 microcontroller. The program controls the on-board LED by flashing it at a fixed rate.



Build and run the project for the source code given below.

```
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#define LED_PIN GPIO_NUM_2          // Define the GPIO pin for the LED
#define ESP_INTR_FLAG_DEFAULT 0
#define Pause 250                    //delay of ms

// Delay for milliseconds
void delay_ms(unsigned int sec){
    vTaskDelay (sec / portTICK_PERIOD_MS);
}

// Function to initialize GPIO
void initialize_gpio()
{
    // Configure GPIOs for LED
    gpio_config_t io_conf_led;
    io_conf_led.intr_type = GPIO_INTR_DISABLE;    // Disable interrupts
    io_conf_led.mode = GPIO_MODE_OUTPUT;          // Set as output mode
    io_conf_led.pin_bit_mask = (1ULL << LED_PIN); // Set the pin mask
    io_conf_led.pull_down_en = GPIO_PULLDOWN_DISABLE; // Disable pull-down resistor
    io_conf_led.pull_up_en = GPIO_PULLUP_DISABLE;    // Disable pull-up resistor
    gpio_config(&io_conf_led);
}

void app_main(void){
    initialize_gpio();
    while (1) {
        gpio_set_level(LED_PIN, 1);    // Turn LED on
        delay_ms(3*Pause);
        gpio_set_level(LED_PIN, 0);    // Turn LED off
        delay_ms(Pause);
    }
}
```

**In-Lab Exercise 1:**

Make changes in the code such that LED blinks with the interval of 100 milliseconds for indefinitely.

**In-Lab Exercise 2:**

Make changes in the code such that LED remains on for 500 milliseconds and remains off for 200 milliseconds indefinitely.

Observe and record the output in the box below.


**Program 2:**

This program demonstrates how to read the logic status of a pushbutton connected with a GPIO of ESP32. Build and run the project for the source code given below.

```
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#define PB_PIN GPIO_NUM_18           // Define the GPIO pin for the Pushbutton
#define ESP_INTR_FLAG_DEFAULT 0

// Function to initialize GPIO
void initialize_gpio(){
    gpio_config_t pb_input;
    // Configure the GPIO as input
    pb_input.intr_type = GPIO_INTR_DISABLE;
    pb_input.mode = GPIO_MODE_INPUT;
    pb_input.pin_bit_mask = (1ULL<<PB_PIN);
    pb_input.pull_down_en = GPIO_PULLDOWN_DISABLE;
    pb_input.pull_up_en = GPIO_PULLUP_DISABLE;
    gpio_config(&pb_input);
}
```

```
// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

//*****
void app_main(void){
    // Initialize GPIO
    initialize_gpio();
    while (1) {
        // Read the state of the GPIO
        int level = gpio_get_level(PB_PIN); //read pushbutton status
    }
}
```

Observe the output and record your observation in the box below.

### Lab Task 1:

Develop a C/C++ code that flashes three LEDs connected with GPIO 18, 19 and 20 with the interval of 250 ms. No more than two LEDs should be on at a time. The task should be executed by the MCU indefinitely. Save your code and attach it to this Lab. Use breadboard to construct the circuit.

### Lab Task 2:

Develop a C/C++ code that controls on-board LED through a pushbutton. LED should be on when pushbutton is pressed and should be off when pushbutton is released. The pushbutton should be connected with GPIO 18. The task should be executed by the MCU indefinitely. Save your code and attach it to this Lab. Use breadboard to construct the circuit.

**Lab Task 3:**

Develop a C/C++ code that controls an LED connected with GPIO4. The brightness of LED should be increasing slowly and then reaching maximum, decrease slowly. The task should be executed by the MCU indefinitely. Save your code and attach it to this Lab. Use breadboard to construct the circuit.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 2

### To Explain the Working of UART of ESP32 and its Programming using C++

#### Objectives

- To understand the RS232 serial communication protocol
- To understand the working of UART on ESP32
- To configure UART on ESP32 for half-duplex communication
- To configure UART on ESP32 for full-duplex communication
- To interface RS232-driven devices with EPS32

#### Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of universal asynchronous receiver/transmitter (UART), RS232 frame format, ESP32 UART programming in C++.

#### RS232 Serial Communication Protocol

RS232 is asynchronous bi-directional serial communication protocol that facilitates interface low data rate devices to exchange data. The hardware that implements the RS232 is called ‘universal asynchronous receiver transmitter’, abbreviated as UART. Many microcontrollers have on-chip UART(s) that can be configured in software to exchange data with RS232-supported devices.

Primarily, a UART has two I/O lines, called RX and TX. The RX line receives data from another device whereas a TX line transmits data to another device. At any time, only two devices can exchange data through UART. Each UART has to be configured for the bit rate at which it will exchange data. The number of total bits that can be either transferred or received by a UART in one second is called ‘Baud rate’, named after French engineer Émile Baudot, who invented the Baudot code in the 1870. Common baud rates used in embedded systems are 2400, 4800, 9600.

RS232 is a byte-oriented LSB-first protocol. This means that the smallest packet that can be either transmitted or received is one byte. The actual data byte is enveloped with one ‘start’ bit, one or two ‘stop’ bit(s) and an optional parity bit. For short range communication, parity bit is often eliminated, and one stop bit is discarded. Figure 2.1 shows the waveform of one data packet that communicates 0x6A byte with one start bit, one stop bit and no parity. This scheme of framing data is called ‘8N1’ frame format. The start bit is always logic 0 and the stop bit is always logic 1.

If baud rate is known, then it can be determined precisely for how long a bit will remain valid. For example, if the baud rate is 9600, then it will take one second to transmit 9600 bits. For 8N1 scheme,

each packet has one start bit, one stop bit and eight data bits. Therefore, 960 packets are communicated in one second. The number of bytes communicated will then be 960 bytes.

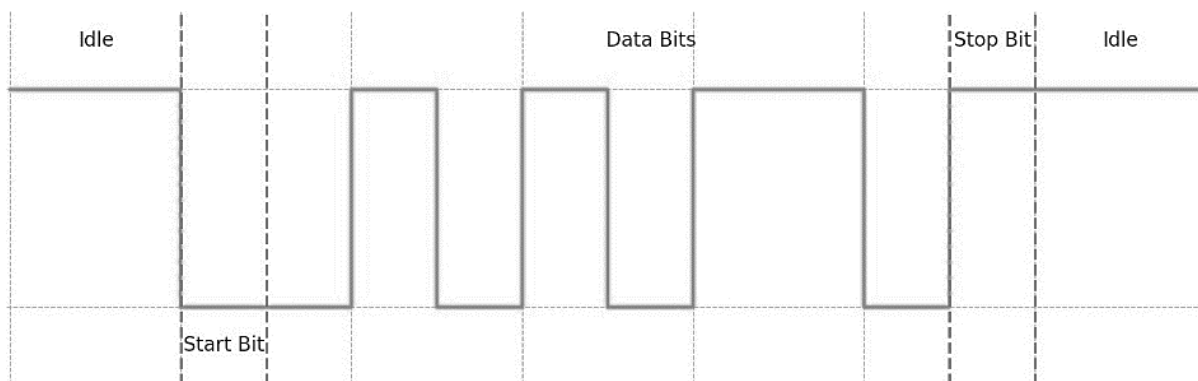


Figure 2.1

## UARTs on ESP32

ESP32 has three UARTs and each can be configured independent of the other. Each UART allows it to be configured in different baud rates and different frame formats. The UART0 has RX0 and TX0 pin designation, UART1 has RX1 and TX1, and UART2 has RX2 and TX2.

### Program 1:

This program demonstrates how to configure UART2 on ESP32 for transmitting the data. Program transmits a byte at a given baud rate with some interval between each transmission. Build and run the project for the source code given below.

```
#include "driver/gpio.h"
#include "driver/uart.h"

#define TXD_PIN GPIO_NUM_17
#define RXD_PIN GPIO_NUM_16

#define UART_NUM UART_NUM_2
#define BAUD 9600

void UART_init(void)
{
    const uart_config_t uart_config = {
        .baud_rate = BAUD,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
```

```

        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    //uart_driver_install(UART_NUM, RX_BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(UART_NUM, &uart_config);
    uart_set_pin(UART_NUM, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
}

void UART_tx(unsigned char byte)
{
    //uint8_t* data = (uint8_t*) malloc(4);
    //data = &byte;
    uart_write_bytes(UART_NUM, &byte, 1);
    //free(data);
}

// Delay for milliseconds
void delay_ms(unsigned int i){
    vTaskDelay (i / portTICK_PERIOD_MS);
}

// Delay for microseconds
void delay_us(unsigned int i){
    esp_rom_delay_us(i);
}

//*****
void app_main(){
    UART_init();

    for(;;){
        UART_tx(0x55);
        delay_ms(10);
    }
}

```

### In-Lab Exercise 1:

Use a breadboard to interface an LED with the TX2 pin of ESP32 and observe its behavior. The LED should flash when data is being transmitted (TXed) from the TX2 pin. Record your observation and note it in the box given below.





### Program 2:

This program demonstrates how to configure the UART2 on ESP32 for receiving the data. Program receives a byte at a given baud rate. Build and run the project for the source code given below.

```
#include "driver/gpio.h"
#include "driver/uart.h"

#define UART_NUM UART_NUM_2
#define TXD_PIN GPIO_NUM_17
#define RXD_PIN GPIO_NUM_16
#define BAUD 9600
static const int RX_BUF_SIZE = 32;
void UART_init(void){
    const uart_config_t uart_config = {
        .baud_rate = BAUD,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    uart_driver_install(UART_NUM, RX_BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(UART_NUM, &uart_config);
    uart_set_pin(UART_NUM, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
}
unsigned char UART_rx(){
    uint8_t data;
    //the portMAX_DELAY waits infinitely for reception
    uart_read_bytes(UART_NUM, &data, 1, portMAX_DELAY);
    return (unsigned char)data;
}
//*****
void app_main(){
    unsigned char in;
```

```

        UART_init();
    for(;;)
        in = UART_rx();
}

```

### In-Lab Exercise 2:

Remove the ESP32 from the computer. Use a breadboard to interface HC05-06 Bluetooth device with ESP32 as given below.

- i. Connect TX of HC05-06 to RX2 of ESP32
- ii. Connect Vcc of HC05-06 to 3.3V pin of ESP32
- iii. Connect GND of HC05-06 to GND of ESP32

Connect the USB cable to the computer. Connect the HC05-06 to your mobile phone through the app. Make changes to the code given in Program 2 such that on-board LED turns on for 1 second when character 'A' is received by the HC05-06. The LED should remain off for any other character received by the HC05-06. Record your observation and note it in the box given below.

### Lab Task 1:

Develop a C/C++ code that flashes an LED connected with GPIO18 for 10 times when any decimal digit is received by the ESP32 through its UART0. Use HC05-06 to send data to ESP32. Save your code and attach it to this Lab. Use breadboard to construct the circuit.

### Lab Task 2:

Develop a C/C++ code that establishes a communication link between two ESP32 boards. One board, called TX, sends a stream of 64 characters to the other ESP32 board, called RX. The stream sent by the TX contains several lower-case letters. The job of the RX is to compute the count of lower-case letters in the stream and store that count in a variable 'count'. The value of the count is to be TXed to the TX board. If the value of count is greater than 32, then on-board LED on TX board should turn on for one second. The on-board LED on TX board should be off otherwise.

Refer to the lab instructor for hardware setup. Save your code and attach it to this Lab. Use breadboard to construct the circuit.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 3

### To Explain the Working of ADC of ESP32 and its Programming using C++

#### Objectives

- To understand the fundamentals of analog to digital converter (ADC)
- To understand the working of on-chip ADC on ESP32
- To configure ADC on ESP32 for single-conversion operation
- To configure ADC on ESP32 for multi-conversion operation
- To Interface analog sensors with ESP32 through ADC

#### Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation ESP32 ADC.

##### ➤ Analog to Digital Conversion

Analog to digital converter (ADC) is an electronic circuit that converts time-varying (analog) voltages into discrete binary levels. Each binary level is assigned a binary code that is the finite length bit pattern, representing the input analog voltage. Each input voltage is represented by a unique binary code. The capability of an ADC to resolve input voltage into binary codes is called its 'resolution' and is represented in 'bits'. Another important parameter of ADC is its 'conversion time', the time an ADC takes to convert input voltage into the output binary code.

An ADC that has 0V to 3V input voltage range and has 10-bit resolution can produce 1024 unique binary codes, linearly distributed over the range of 0V to 3V. In this case, the output binary value for 0V input will be '0000000000' whereas for 3V this will be '1111111111'. Therefore, the variation of 2.9 mV will cause minimum variation in the output code.

It can be inferred from this that an ADC with higher resolution will be more sensitive to input voltage variations. However, increasing resolution of an ADC is paid by its slow conversion speed. Higher the resolution, higher will be conversion time.

##### ➤ ADCs on ESP32

ESP32 has two successive approximation ADCs named ADC1 and ADC2. ADC1 is a 12-bit ADC and has 8 channels. Each channel input is accessible through GPIO32 to GPIO39, successively. Recall that when these pins are configured as analog inputs of ADC1, they are no longer accessible as digital GPIOs. ADC2 has 10 channels that are mapped on GPIO0, GPIO2, GPIO4, GPIO12 to

GPIO15, and GPIO25 to GPIO27. Each ADC can be configured at different attenuation levels. Attenuation levels define the input analog voltage range that are to be converted into the digital value. Different attenuation levels and corresponding input voltage range are given in Table 3.1.

Table 3.1

Attenuation macros	Input voltage range
ADC_ATTEN_DB_0	100 mV ~ 950 mV
ADC_ATTEN_DB_2_5	100 mV ~ 1250 mV
ADC_ATTEN_DB_6	150 mV ~ 1750 mV
ADC_ATTEN_DB_11	150 mV ~ 2450 mV

### Program 1:

This program demonstrates how to configure the ADC1 to read a conversion through one of its 8 channels. Build and run the project for the source code given below.

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/adc.h"
#include "esp_adc_cal.h"

//ADC settings
#define DEFAULT_VREF 1100 // Default reference voltage (in mV) for ESP32
#define NO_OF_SAMPLES 8 // Multisampling
#define ADC_CHANNEL ADC1_CHANNEL_6 // ADC1 channel connected to GPIO34

// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

//function to read average value of multi-conversion result
uint16_t ADC_result(uint8_t samples){
    uint16_t adc = 0;
    for (uint8_t i = 0; i < samples; i++) {
        adc += adc1_get_raw(ADC_CHANNEL);
    }
    adc /= samples; // Average the samples
```

```

    return(adc);
}

/*****
void app_main() {
    uint16_t ADC = 0;

    // Initialize the ADC
    adc1_config_width(ADC_WIDTH_BIT_12); // 12-bit ADC width
    adc1_config_channel_atten(ADC_CHANNEL, ADC_ATTEN_DB_12);

    for(;;){
        ADC = ADC_result(8);
        delay_ms(200);
    }
}

```

### In-Lab Exercise 1:

Make changes to Program 1 such that the on-board LED turns on when ADC output is in the range of 1000 to 2500. Use a breadboard and a potentiometer to implement the circuit. Show your work to the instructor and record your observations in the box below.

#### *Safety Instructions:*

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with computer when circuit has been fully constructed.

### Lab Task 1:

Develop a C/C++ code that interfaces LM35 temperature sensor with ESP32 through its ADC2. The program should have a software pre-set temperature. On-board LED should turn on when measured temperature exceeds the pre-set value. The LED should remain off otherwise.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

**Lab Task 2:**

Extend the C/C++ code developed for Lab Task 1 to get the pre-set value through channel 1 of ADC1. Interface LM35 with channel 2 of ADC 1. Set the temperature through channel 1 and when temperature exceeds the pre-set value, turn on the on-board LED.

Make use of multiple conversions of LM35 readings and take average of these readings.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.



## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 4

### To Follow the I2C Protocol and Interfacing LCD with ESP32 using C++

#### Objectives

- To understand the fundamentals of inter-integrated circuit (I2C) protocol
- To understand the I2C programming of ESP32
- To interface liquid crystal display (LCD) with ESP32 using I2C protocol
- To control LCD through software to display text

#### Pre-Lab Exercise

Read the details given below in order to comprehend the principle of inter-integrated circuits (I2C) protocol and by using it, interfacing an LCD with ESP32. In addition, read the details for basic commands of controlling LCD operation.

#### Inter-integrated Circuit Protocol

I2C is a synchronous serial protocol that allows data communication among several electronic devices. This protocol uses two lines, called clock (SCL) and data (SDA) that carry binary signals to allow bidirectional data transfer. The SCL line transfers clock signals and SDA line transfers data among I2C devices.

The SCL and SDA lines are open-drain lines that need pull-up resistors to establish proper logic level on these lines. I2C devices can establish a network that is governed by one device called 'master'. Other devices in the network which are connected with the master are called slaves. In an I2C network, there is only master. However, there can be several slaves that can range from 1 to 127. Figure 4.1 shows I2C data transfer waveforms.

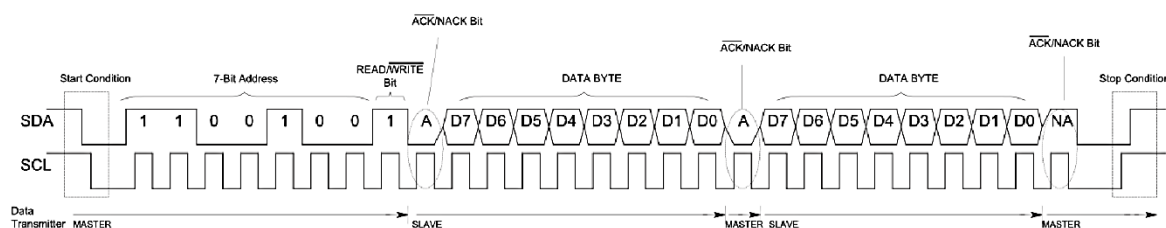


Figure 4.1

The hardware architecture of connecting slaves with master using SDA and SCL line with pull-up resistors is shown in Figure 4.2. It can be inferred from Figure 2 that each slave has a unique address.

This uniqueness of addresses in connected devices is the key to identify a particular slave to establish data communication. These slave addresses are always generated by the master. The master has 0x00 address. Generally,  $V_{cc}$  stays at 5V for most of the I2C networks. If slaves have different voltage levels for their SCL and SDA lines, then voltage converters are to be deployed. I2C protocol supports two clock frequencies, the slow-speed 100 kHz and high-speed 400 kHz.

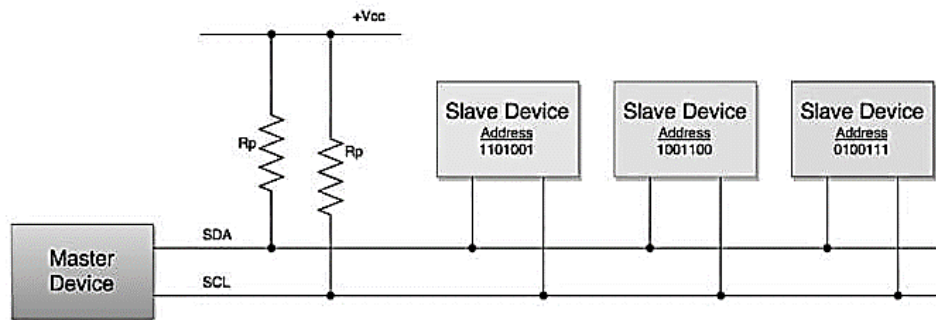


Figure 4.2

Several devices such as ADCs, DACs, sensors, real-time clocks, and many peripheral devices have I2C interface for data exchange.

#### ➤ I2C on ESP32

ESP32 has on-chip I2C interface that allows communication with other I2C devices. GPIO21 and GPIO22 are used for SDA and SCL, respectively. Each of these GPIOs must be pulled up in software to allow the ESP32 to communicate with slaves.

#### ➤ Liquid Crystel Display (LCD)

LCD is one of the most widely used output device in embedded systems. An LCD has a display controller that controls the functionality of the LCD such as the data to be printed on it, processing the command to control the communication modes with LCD, its cursor position and other features. Hitachi's 44780 is the commonly used LCD controller. Display panel of LCD comes in several sizes such as 16x2, 16x4, 20x4 etc. For example, a 16x2 panel size display can show 32 characters. These are arranged as a 16x2 matrix with 16 columns and 2 rows. Figure 4.3 shows a picture of 20x4 LCD with some text written on it.

These LCD units operate on 5V DC and accept TTL logic levels for data exchange. In order to display text on LCD, it has to be configured first. Configuring the LCD consists of several steps that involve data transfer to it by the driver controller, generally a microcontroller. An LCD unit has parallel interface to transfer one byte data to it. In case where driver MCU does not have these many control lines, a serial to parallel converter is needed. PCF8574 is a commonly used converter for this purpose that has I2C interface for accepting serial data and provides 8-bit parallel data at its output.



Figure 4.3

An LCD with a 44780 controller has 14 or 16 connectors for power, control and data transfer. These connectors and their functionality are given in Table 4.1.

**Table 4.1**

Pin No.	Pin Name	Pin Type	Pin Description	Pin Connection
Pin 1	Ground	Source Pin	This is a ground pin of LCD	Connected to the ground of the MCU/ Power source
Pin 2	VCC	Source Pin	This is the supply voltage pin of LCD	Connected to the supply pin of Power source
Pin 3	V0/VEE	Control Pin	Adjusts the contrast of the LCD.	Connected to a variable POT that can source 0-5V
Pin 4	Register Select	Control Pin	Toggles between Command/Data Register	Connected to a MCU pin and gets either 0 or 1. 0 -> Command Mode 1-> Data Mode
Pin 5	Read/W rite	Control Pin	Toggles the LCD between Read/Write Operation	Connected to a MCU pin and gets either 0 or 1. 0 -> Write Operation 1-> Read Operation
Pin 6	Enable	Control Pin	Must be held high to perform Read/Write Operation	Connected to MCU and always held high.
Pin 7-14	Data Bits (0-7)	Data/Command Pin	Pins used to send Command or data to the LCD.	<u>In 4-Wire Mode</u> Only 4 pins (0-3) are connected to MCU <u>In 8-Wire Mode</u> All 8 pins (0-7) are connected to MCU

Pin 15	LED Positive	LED Pin	Normal LED like operation to illuminate the LCD	Connected to +5V
Pin 16	LED Negative	LED Pin	Normal LED like operation to illuminate the LCD connected with GND.	Connected to ground

### LCD Commands

LCD controller has a rich set of command that facilitate user to control the functionality of LCD. Table 4.2 provides the basic list of commands that are frequently used while working with LCDs.

**Table 4.2**

Sr. No.	Hex Code	Command to LCD instruction Register
1	01	Clear display screen
2	02	Return home
3	04	Decrement cursor (shift cursor to left)
4	06	Increment cursor (shift cursor to right)
5	05	Shift display right
6	07	Shift display left
7	08	Display off, cursor off
8	0A	Display off, cursor on
9	0C	Display on, cursor off
10	0E	Display on, cursor blinking
11	0F	Display on, cursor blinking
12	10	Shift cursor position to left
13	14	Shift the cursor position to the right
14	18	Shift the entire display to the left
15	1C	Shift the entire display to the right
16	80	Force cursor to the beginning (1st line)
17	C0	Force cursor to the beginning (2nd line)
18	38	2 lines and 5×7 matrix

**Program 1:**

This program demonstrates how to configure PCF8574 using I2C protocol to display text on an LCD.

```
#include <stdio.h>
#include "driver/i2c.h"

// I2C Configuration
#define I2C_MASTER_SCL_IO 22      // GPIO number for I2C master clock
#define I2C_MASTER_SDA_IO 21      // GPIO number for I2C master data
#define I2C_MASTER_NUM I2C_NUM_0 // I2C port number for master
#define I2C_MASTER_FREQ_HZ 400000 // I2C master clock frequency
#define I2C_MASTER_TX_BUF_DISABLE 0 // I2C master does not need buffer
#define I2C_MASTER_RX_BUF_DISABLE 0 // I2C master does not need buffer

#define cmd_delay 500              //ms delay for command processing time
#define pulse_delay 10             //ms delay for signal active time

// I2C Address of the PCF8574
#define PCF8574_ADDR 0x27

// LCD Commands
#define LCD_CMD_CLEAR_DISPLAY 0x01
#define LCD_CMD_RETURN_HOME 0x02
#define LCD_CMD_ENTRY_MODE 0x06
#define LCD_CMD_DISPLAY_ON 0x0C
#define LCD_CMD_FUNCTION_SET 0x28 // 4-bit mode, 2-line, 5x7 dots

// LCD pins controlled by PCF8574
#define LCD_EN 0x04 // Enable bit
#define LCD_RW 0x02 // Read/Write bit (always 0 for writing)
#define LCD_RS 0x01 // Register select bit

// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}
```

// Function to initialize the I2C interface

```
static void i2c_master_init() {
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = I2C_MASTER_SDA_IO,
        .scl_io_num = I2C_MASTER_SCL_IO,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = I2C_MASTER_FREQ_HZ,
    };

    i2c_param_config(I2C_MASTER_NUM, &conf);
    i2c_driver_install(I2C_MASTER_NUM, conf.mode,
                      I2C_MASTER_RX_BUF_DISABLE,
                      I2C_MASTER_TX_BUF_DISABLE, 0);
}
```

// Function to send a byte to the PCF8574 (for controlling the LCD)

```
void pcf8574_write(uint8_t data) {
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (PCF8574_ADDR << 1) | I2C_MASTER_WRITE, true);
    i2c_master_write_byte(cmd, data, true);
    i2c_master_stop(cmd);
    i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, cmd_delay / portTICK_PERIOD_MS);
    i2c_cmd_link_delete(cmd);
}
```

// Send an enable pulse to the LCD

```
void lcd_send_enable_pulse(uint8_t data) {
    pcf8574_write(data | LCD_EN); // Enable high
    vTaskDelay (pulse_delay / portTICK_PERIOD_MS);
    pcf8574_write(data & ~LCD_EN); // Enable low
    vTaskDelay (pulse_delay / portTICK_PERIOD_MS);
}
```

```

// Send a command or data to the LCD
void lcd_send(uint8_t value, bool isData) {
    uint8_t high_nibble = (value & 0xF0);    // High nibble
    uint8_t low_nibble = ((value << 4) & 0xF0); // Low nibble

    if (isData) {
        high_nibble |= LCD_RS;
        low_nibble |= LCD_RS;
    }

    // Send high nibble
    lcd_send_enable_pulse(high_nibble);

    // Send low nibble
    lcd_send_enable_pulse(low_nibble);
}

// Send a command to the LCD
void lcd_command(uint8_t command) {
    lcd_send(command, false);
}

// Send data to the LCD
void lcd_data(uint8_t data) {
    lcd_send(data, true);
}

// Initialize the LCD
void lcd_init() {
    delay_ms(500);    // Wait for LCD to power up
    lcd_command(LCD_CMD_FUNCTION_SET); // 4-bit mode, 2 lines, 5x7 dots
    lcd_command(LCD_CMD_DISPLAY_ON);    // Display on, cursor off
    lcd_command(LCD_CMD_ENTRY_MODE);    // Entry mode set
    lcd_command(LCD_CMD_RETURN_HOME);    // Return home
    lcd_command(LCD_CMD_CLEAR_DISPLAY); // Clear display
}

```



```

//function sets the cursor position
void cursor_position(char r, char c){
    if(r==0)
        lcd_command(0x80+c);
    if(r==1)
        lcd_command(0x80+c+0x40);
} //end cursor_position

//string write function
void lcd_print(char *dat){
    //The # operator turns the argument it precedes into a quoted string.
    #define mkstr(dat) #dat
    while(*dat != '\0')
        lcd_data(*(dat++)); //send single character to LCD
} //end lcd_print

/*****
void app_main() {
    char buff[20];

    // Initialize I2C master
    i2c_master_init();
    lcd_init();

    cursor_position(0,0);
    lcd_print("ESP32 ADC & LCD");
    cursor_position(1,0);
    sprintf(buff,"HELLO, FROM CUI");
    lcd_print(buff);

    lcd_command(0x0C); //turn cursor off

    while(1);
}

```

Following safety instruction must be observed carefully while setting up the circuit.

**Safety Instructions:**

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with computer when circuit has been fully constructed.

**In-Lab Exercise 1:**

Make changes to Program 1 such that the following text appears on LCD.

On upper line:     Your Name

On lower line:     Your Reg. No.

Use a breadboard to implement the circuit. Show your work to the instructor and record your observations in the box below.

**In-Lab Exercise 2:**

Make changes to Program 1 such that a counting from 0 to 100 is displayed on LCD. Count should be updated by with interval of 500 ms. LCD should display the counting as given below.

On upper line:     Count (dec.) = 005

On lower line:     Count (hex.) = 005

Use a breadboard to implement the circuit. Show your work to the instructor and record your observations in the box below.

**Lab Task 1:**

Develop a C/C++ code that reads data from on-chip ADC of ESP32 and displays the value on LCD.

Use appropriate text to display the ADC output.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

### Lab Task 2:

Extend the C/C++ code developed for Lab Task 1 to implement a thermostat using ESP32 MCU. The temperature is to be sensed using LM35. On-board LED should turn on when measured temperature exceeds the pre-set temperature value. Add an LCD to your circuit that displays the current temperature on upper line and pre-set temperature on lower line.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

### Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 5

### To Construct C++ Code for I2C Link to Interface RTC and LCD with ESP32

#### Objectives

- To understand the software implementation of I2C on ESP32 using GPIOs
- To understand file handling and managing software development for an embedded system
- To interface multiple devices with ESP32 through I2C
- To understand real-time data acquisition and timestamping

#### Pre-Lab Exercise

Read the details given below in order to comprehend the implementation of I2C on ESP32 using C++ code. Also, understand the file handling and managing embedded software development.

#### Software implementation of I2C using GPIOs

I2C is a serial communication protocol that allows a microcontroller (or any digital controller) to connect sensors, RTCs, displays and several other digital devices using two IO lines. Figure 5.1 shows the waveforms of SDA and SCL lines to communicate a 3-byte message from the master to the slave.

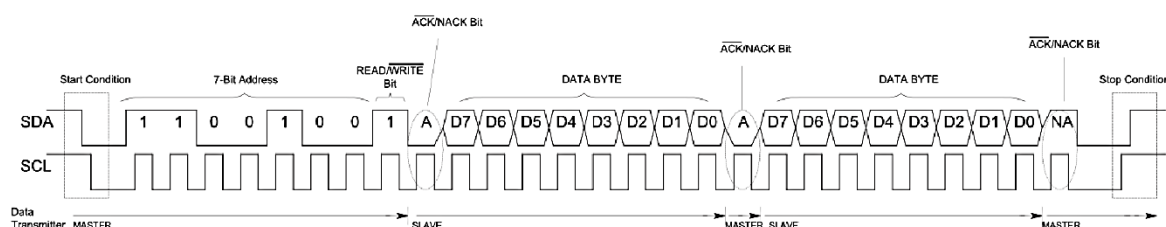


Figure 5.1

The nature of I2C protocol is simple and this simplicity allows us to use two GPIOs on ESP32 to work as SDA and SCL lines. This method of implementing the I2C using GPIOs serves as a software blueprint for those processors or microcontrollers that are not equipped with an I2C channel. Moreover, this approach allows us to virtually implement multiple I2C channels on a single embedded controller such as ESP32.

As a first example, software I2C is implemented to communicate with PCF8574 I2C to parallel converter that drives a 20x4 LCD. The C/C++ program given in Program 1 serves as blueprint to create a channel on ESP32 for I2C communication. This program prints some text on the LCD that is

connected with PCF8574, whereas PCF8574 is interfaced with ESP32 through software-implemented I2C channel.

Run the program given in Program 1 and show the output to the lab instructor. Make use of breadboard to implement the circuit. Follow the safety conditions.

***Safety Instructions:***

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

**Program 1:**

This program has a dependency file *PCF8574\_LCD.c* that contains the PCF8574 interfacing and an LCD control function using software I2C. The objective of this exercise is to understand how to manage different software parts at the development phase of an embedded system.

Copy the following code into the *main.c* file of your project.

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "PCF8574_LCD.c"

// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

/*****
void app_main() {
    char buff[20];

    PCF8574_LCD_init();

    cursor_position(0,0);
    lcd_print("ABCDEFGHJKLMNOPQRST");
    cursor_position(1,0);
    lcd_print("01234567890123456789");
    cursor_position(2,0);
    lcd_print("abcdefghijklmnopqrst");
    cursor_position(3,0);
    lcd_print("98765432109876543210");
}
```

```

        for(;;);
    }

```

Create a new file with the name PCF8574\_LCD.c and copy the following code into this file. Save the file and place it in the 'main' folder of your project.

```

/*****

```

This .c file contains the I2C implementation for PCF8574 and 44780 LCD functions to print ASCII data on LCD.

```

CPU:                ESP32
Written By:         Usman Rafique
Dated:              Oct. 04, 2024      Friday

```

```

*****/

```

```

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

```

```

// Define the GPIO pins
#define PCF8574_I2C_SCL 18
#define PCF8574_I2C_SDA 19

```

```

#define cmd_delay 20          //ms delay for command processing time
#define pulse_delay 10        //ms delay for signal active time
#define PCF8574_I2C_DELAY 10  // microseconds delay for timing adjustment

```

```

// I2C Address of the PCF8574
#define PCF8574_ADDR 0x4E

```

```

// LCD pins controlled by PCF8574
#define LCD_EN 0x04 // Enable bit
#define LCD_RW 0x02 // Read/Write bit (always 0 for writing)
#define LCD_RS 0x01 // Register select bit

```

```

// Utility functions for I2C bit-banging
void PCF8574_i2c_delay() {
    esp_rom_delay_us(PCF8574_I2C_DELAY);
}

```

```
// I2C initialization for PCF8574.
```

```
void PCF8574_i2c_init() {
    gpio_set_direction(PCF8574_I2C_SCL, GPIO_MODE_OUTPUT);
    gpio_set_direction(PCF8574_I2C_SDA, GPIO_MODE_OUTPUT);
    gpio_set_level(PCF8574_I2C_SCL, 1);
    gpio_set_level(PCF8574_I2C_SDA, 1);
    gpio_pullup_en(PCF8574_I2C_SCL);
    gpio_pullup_en(PCF8574_I2C_SDA);
}
```

```
void PCF8574_i2c_start() {
    gpio_set_level(PCF8574_I2C_SDA, 1);
    gpio_set_level(PCF8574_I2C_SCL, 1);
    PCF8574_i2c_delay();
    gpio_set_level(PCF8574_I2C_SDA, 0); // Start condition
    PCF8574_i2c_delay();
    gpio_set_level(PCF8574_I2C_SCL, 0);
}
```

```
void PCF8574_i2c_stop() {
    gpio_set_level(PCF8574_I2C_SDA, 0);
    gpio_set_level(PCF8574_I2C_SCL, 1);
    PCF8574_i2c_delay();
    gpio_set_level(PCF8574_I2C_SDA, 1); // Stop condition
    PCF8574_i2c_delay();
}
```

```
void PCF8574_i2c_write_bit(uint8_t bit) {
    gpio_set_level(PCF8574_I2C_SDA, bit);
    PCF8574_i2c_delay();
    gpio_set_level(PCF8574_I2C_SCL, 1);
    PCF8574_i2c_delay();
    gpio_set_level(PCF8574_I2C_SCL, 0);
}
```

```
uint8_t PCF8574_i2c_read_bit() {
    gpio_set_level(PCF8574_I2C_SDA, 1); // Release SDA to allow slave to control it
    PCF8574_i2c_delay();
    gpio_set_level(PCF8574_I2C_SCL, 1);
    PCF8574_i2c_delay();
```

```

uint8_t bit = gpio_get_level(PCF8574_I2C_SDA);
gpio_set_level(PCF8574_I2C_SCL, 0);
return bit;
}

void PCF8574_i2c_write_byte(uint8_t byte) {
    for (int i = 0; i < 8; i++) {
        PCF8574_i2c_write_bit((byte & 0x80) != 0); // Write MSB first
        byte <<= 1;
    }
    // Read ACK bit
    PCF8574_i2c_read_bit();
}

// Function to send a byte to the PCF8574 (for controlling the LCD)
void PCF8574_write(uint8_t data) {
    PCF8574_i2c_start();
    PCF8574_i2c_write_byte(PCF8574_ADDR);
    PCF8574_i2c_write_byte(data);
    PCF8574_i2c_stop();
}

// Send an enable pulse to the LCD
void lcd_send_enable_pulse(uint8_t data) {
    PCF8574_write(data | LCD_EN); // Enable high
    esp_rom_delay_us(100);
    PCF8574_write(data & ~LCD_EN); // Enable low
    esp_rom_delay_us(100);
}

// Send a command or data to the LCD
void lcd_send(uint8_t value, bool isData) {
    uint8_t high_nibble = (value & 0xF0); // High nibble
    uint8_t low_nibble = ((value << 4) & 0xF0); // Low nibble

    if (isData) {
        high_nibble |= LCD_RS;
        low_nibble |= LCD_RS;
    }

    // Send high nibble

```



```

        lcd_send_enable_pulse(high_nibble);
        // Send low nibble
        lcd_send_enable_pulse(low_nibble);
    }

// Send a command to the LCD
void lcd_command(uint8_t command) {
    lcd_send(command, false);
}

// Send data to the LCD
void lcd_data(uint8_t data) {
    lcd_send(data, true);
}

// Initialize the LCD
void lcd_init() {
    vTaskDelay(pdMS_TO_TICKS(1000));    // Wait for LCD to power up
    lcd_command(0x28);                  // 4-bit mode, 2 lines, 5x7 dots
    lcd_command(0x0C);                  // display on, cursor off
    lcd_command(0x06); // cursor auto-right move
    lcd_command(0x02);                  // cursor at home
    lcd_command(0x01);                  // clear display
    vTaskDelay(pdMS_TO_TICKS(200));
}

//function sets the cursor position
void cursor_position(char r, char c){
    if(r==0)    //row 0
        lcd_command(0x80+c);
    if(r==1)    //row 1
        lcd_command(0xC0+c);
    if(r==2)    //row 2
        lcd_command(0x94+c);
    if(r==3)    //row 3
        lcd_command(0xD4+c);
} //end cursor_position

//string write function
void lcd_print(char *dat){
//The # operator turns the argument it precedes into a quoted string.

```

```

#define mkstr(dat) #dat
        while(*dat != '\0')
            lcd_data(*(dat++));    //send single character to LCD
    }        //end lcd_print

//function configures PCF8574 to communicate with LCD. Must be called in main()
void PCF8574_LCD_init(){
    PCF8574_i2c_init();
    lcd_init();
}

```

### In-Lab Exercise 1:

Make changes in Program 1 such that it uses other GPIOs than those used in Program 1 for implementing I2C channel.

Construct the circuit and show the output to the lab instructor.

### In-Lab Exercise 2:

Make changes in Program 1 such that it implements two software I2C channels on ESP32. Interface two LCDs and show some text on each LCD.

Construct the circuit and show the output to the lab instructor.

### Interfacing Real-Time Clock (RTC)

A real-time clock (RTC) is a digital circuit that automatically computes the time, date and year. Such a circuit is DS1307 RTC that has I2C interface and provides features of implementing a complete time management system. DS1307 has a 64x8 volatile memory that can be written to and read from via I2C interface. Memory locations from 0 to 7 store timing information which are updated automatically. The information stored in these locations is shown in Figure 5.2.

ADDRESS	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00-59
01H	0	10 Minutes			Minutes				Minutes	00-59
02H	0	12	10 Hour	10 Hour	Hours				Hours	1-12 +AM/PM 00-23
		24	PM/AM							
03H	0	0	0	0	0	DAY			Day	01-07
04H	0	0	10 Date		Date				Date	01-31
05H	0	0	0	10 Month	Month				Month	01-12
06H	10 Year				Year				Year	00-99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	---
08H-3FH									RAM 56 x 8	00H-FFH

Figure 5.2

The memory locations from 08H (8) to 3FH (63) are general-purpose read/write locations.

### Program 2:

This program has a dependency file *DS1307.c* that contains DS1307 interfacing functions using software I2C.

Copy the following code into the *main.c* file of your project.

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "PCF8574_LCD.c"
#include "DS1307.c"
#include <stdio.h>
```

```
typedef unsigned char UCHAR;
typedef unsigned int UINT;
```

```
//RTC time and date variables
UCHAR hrs,min,sec;
```

```
// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}
```

```
//function to convert decimal number to BCD format
UCHAR dec2bcd(UCHAR num){
    return(((num/10)*6)+num);
} //end dec2bcd
```

```
//function to set new time, day and date in DS1307
void Set_Time(UCHAR h,UCHAR m,UCHAR s){
```

```
    WR_DS1307(0,dec2bcd(s)&0x7F); //write seconds with CH=0
    WR_DS1307(2,dec2bcd(h)); //write hours
} //end Set_Time
```

```
//function to get updated time from DS1307
void Get_Time(){
```

WR\_DS1

```

        sec = RD_DS1307(0);
        min = RD_DS1307(1);
        hrs = RD_DS1307(2);
    } //end Get_Time

//function displays time, day and date on 44780 LCD
void Display_Time(){
    lcd_data((hrs>>4)+48);
    lcd_data((hrs&0x0F)+48);
    lcd_data(':');

    lcd_data((min>>4)+48);
    lcd_data((min&0x0F)+48);
    lcd_data(':');

    lcd_data((sec>>4)+48);
    lcd_data((sec&0x0F)+48);

} //end Display_Time

/*****
void app_main() {
    //UCHAR buff[20];

    PCF8574_LCD_init();
    DS1307_i2c_init();
    cursor_position(0,3);
    lcd_print("ESP32 & DS1307");
    delay_ms(500);

    Set_Time(23,59,47); //24H time format

    for(;;){
        Get_Time();
        cursor_position(1,4);
        Display_Time();
        delay_ms(1000);
    } //end for
}

```

Make a file 'DS1307.c' that contains the following code. Place this file in 'main' folder.

```
/******
```

This file interfaces DS1307 RTC with ESP32 using software I2C.

Created By: Usman Rafique

Dated: Oct. 16, 2024

```
*****/
```

```
#define DS1307_I2C_DELAY 10 // microseconds delay for timing adjustment
```

```
// Define the GPIO pins
```

```
#define DS1307_I2C_SCL 22
```

```
#define DS1307_I2C_SDA 23
```

```
// Write Address of DS1307
```

```
#define DS1307_ADDR_W 0xD0
```

```
// Read Address of DS1307
```

```
#define DS1307_ADDR_R 0xD1
```

```
#define ACK 1 //ACK signal
```

```
#define NACK 0 //NACK signal
```

```
// Utility functions for I2C bit-banging
```

```
void DS1307_i2c_delay() {
```

```
    esp_rom_delay_us(DS1307_I2C_DELAY);
```

```
}
```

```
// I2C initialization function for DS1307
```

```
void DS1307_i2c_init() {
```

```
    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
```

```
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_OUTPUT);
```

```
    gpio_set_level(DS1307_I2C_SCL, 1);
```

```
    gpio_set_level(DS1307_I2C_SDA, 1);
```

```
    gpio_pullup_dis(DS1307_I2C_SCL);
```

```
    gpio_pullup_dis(DS1307_I2C_SDA);
```

```
}
```

```
void DS1307_i2c_start() {
```

```
    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
```

```
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_OUTPUT);
```

```
    gpio_set_level(DS1307_I2C_SDA, 1);
```

```

        gpio_set_level(DS1307_I2C_SCL, 1);
        DS1307_i2c_delay();
        gpio_set_level(DS1307_I2C_SDA, 0); // Start condition
        DS1307_i2c_delay();
        gpio_set_level(DS1307_I2C_SCL, 0);
    }

void DS1307_i2c_stop() {
    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_OUTPUT);
    gpio_set_level(DS1307_I2C_SDA, 0);
    gpio_set_level(DS1307_I2C_SCL, 1);
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SDA, 1); // Stop condition
    DS1307_i2c_delay();
}

//function generates I2C ACK
void DS1307_i2c_ack(){
    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_OUTPUT);
    gpio_set_level(DS1307_I2C_SDA, 0);
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SCL, 1);
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SCL, 0);
    DS1307_i2c_delay();
}    //end DS1307_i2c_ack

//function generates I2C NACK
void DS1307_i2c_nack(){
    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_OUTPUT);
    gpio_set_level(DS1307_I2C_SDA, 1);
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SCL, 1);
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SCL, 0);
    DS1307_i2c_delay();
}    //end DS1307_i2c_nack

```

```

//function to write an I2C packet
void DS1307_I2C_Write(uint8_t data){
    uint8_t i;

    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_OUTPUT);

    gpio_set_level(DS1307_I2C_SCL, 0);
    DS1307_i2c_delay();

    for (i = 0; i < 8; i++) {
        if (data & 0x80) {
            gpio_set_level(DS1307_I2C_SDA, 1);
            DS1307_i2c_delay();
        }
        else {
            gpio_set_level(DS1307_I2C_SDA, 0);
            DS1307_i2c_delay();
        }

        gpio_set_level(DS1307_I2C_SCL, 1);
        DS1307_i2c_delay();

        DS1307_i2c_delay();

        data <<= 1;
    } //end for
    gpio_set_level(DS1307_I2C_SCL, 0);
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SDA, 0);
    DS1307_i2c_delay();

    //ignore ack from salve
    gpio_set_level(DS1307_I2C_SCL, 1);
    DS1307_i2c_delay();
    DS1307_i2c_delay();
    DS1307_i2c_delay();
    gpio_set_level(DS1307_I2C_SCL, 0);
} //end DS1307_I2C_Write
//function to read an I2C packet

```

```

uint8_t DS1307_I2C_Read(uint8_t j){
    uint8_t i,in,byte;

    in = 0;
    byte = 0;

    gpio_set_direction(DS1307_I2C_SCL, GPIO_MODE_OUTPUT);
    gpio_set_direction(DS1307_I2C_SDA, GPIO_MODE_INPUT);
    gpio_set_level(DS1307_I2C_SCL, 0);
    DS1307_i2c_delay();

    for(i=0; i<8; i++){
        gpio_set_level(DS1307_I2C_SCL, 1);
        DS1307_i2c_delay();
        byte <<= 1;
        in = gpio_get_level(DS1307_I2C_SDA);
        byte |= in;

        DS1307_i2c_delay();
    } //end for

    //send ACK if j = 1
    if(j){
        DS1307_i2c_ack();
    } //end if
    else{
        DS1307_i2c_nack();
    } //end else

    return byte;
} //end DS1307_I2C_Read
//Function to write to any location of DS1307
void WR_DS1307(uint8_t add, uint8_t byte){
    DS1307_i2c_start();
    DS1307_I2C_Write(DS1307_ADDR_W);
    DS1307_I2C_Write(add);
    DS1307_I2C_Write(byte);
    DS1307_i2c_stop();
} //end WR_DS1307
//Function to read from any location of DS1307

```



```

uint8_t RD_DS1307(uint8_t add){
    uint8_t out;

    DS1307_i2c_start();
    DS1307_I2C_Write(DS1307_ADDR_W);
    DS1307_I2C_Write(add);
    DS1307_i2c_start();
    DS1307_I2C_Write(DS1307_ADDR_R);

    out = DS1307_I2C_Read(NACK);

    return out;
} //end RD_DS1307

```

**Lab Task 1:**

Develop a C/C++ code that interfaces DS1307 RTC with ESP32 using dedicated I2C channel and LCD using software I2C. The program should display the following information on LCD according to the following format.

- On line 1:     Display the time
- On line 2:     Display name of the day
- On line 3:     Display date and name of the month
- On line 4:     Display year

**Lab Task 2:**

Develop a C/C++ code that interfaces DS1307 RTC with ESP32 using dedicated I2C channel and LCD using software I2C. The program should display the following information on LCD according to the following format.

- On line 1:     Display the time
- On line 2:     Display name of the day
- On line 3:     Display date and name of the month
- On line 4:     Display year

The program should set a soft-coded alarm (with seconds, minutes, hours, day, date, month and year) and when time matches the alarm data, on board LED should flash.

The program should also include a pushbutton. Alarm data should be displayed on LCD for one second when pushbutton is pressed.

**Lab Task 3:**

Develop a C/C++ code that interfaces DS1307 RTC, an LCD and an LM35 temperature sensor with ESP32. Code should sample the LM35 reading after every second and after timestamping it, display on LCD.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_

**Date:** \_\_\_\_\_

## LAB # 6

### To Construct C++ Code for SPI Interface and Matrix Keypad

#### Objectives

- To understand Serial Peripheral Interface (SPI) protocol communication
- To interface SPI devices with ESP32
- To interface matrix keypad with ESP32

#### Pre-Lab Exercise

Read the details below to comprehend the SPI protocol structure and by applying this protocol, interface SPI devices with ESP32. In addition, also understand how a matrix keypad can be scanned and interfaced with ESP32.

#### SPI protocol

SPI is a widely used serial communication protocol that facilitates bidirectional binary data communication among a digital controller and peripheral devices such as ADCs, DACs, EEPROMs, SRAMs and digital displays. SPI is a synchronous communication protocol that primarily uses four digital lines. These lines are CLK (clock), CS (chip select), MISO (data input, from peripheral (slave) to the controller (master)), and MOSI (data out, from controller (master) to the peripheral (slave)). MISO and MOSI are also called DIN and DOUT, respectively. CLK is the clock signal that is controlled by the master. SPI can operate safely from 100 kHz to 4 MHz.

CS is the chip select line that enables that slave device with which master wants to communicate. If more than one slaves are to be interfaced with single master, then an  $N \times 2^N$  decoder is to be used. MISO is the master input line that transfers data from the slave to the master. MOSI is the master output line that transfers data from the master to the slave.

SPI is a high-speed link that facilitates hardware designers interfacing several devices of different nature in an embedded system. Owing to its simplicity, it can be easily programmed in a general-purpose microcontroller using its GPIOs. It is a one-byte or two-byte oriented protocol that uses MSB-first scheme for data communication. Data is sampled on the rising edge of the clock and is shifted to either master or slave on the falling edge. During entire communication session, CS remains at the active logic level.

Figure 6.1 shows the schematic diagram of interfacing multiple peripherals with a master using SPI link.

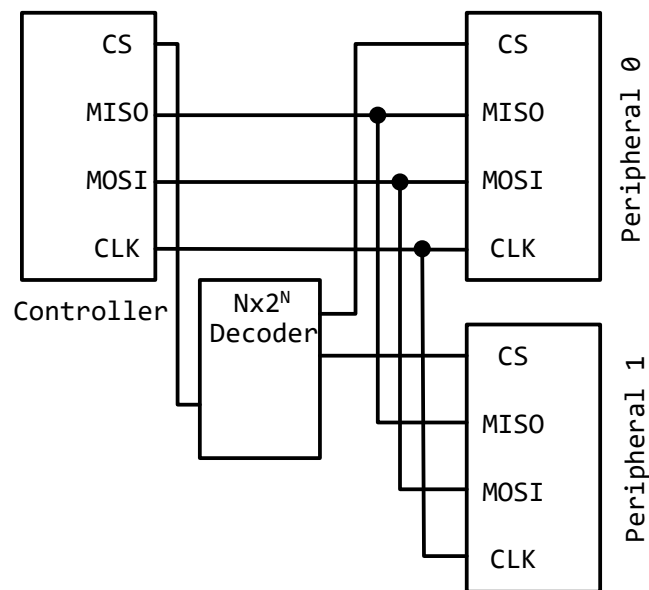


Figure 6.1

Figure 6.2 shows the waveforms of SPI communication link.

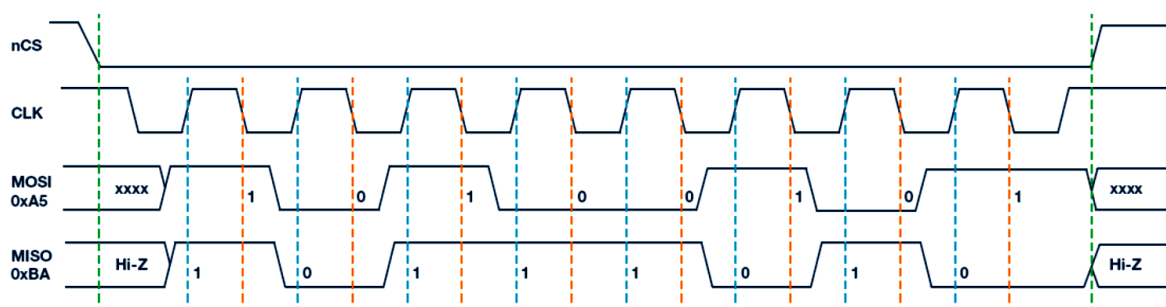


Figure 6.2

### ➤ MAX7219 8-digit multiplexed LED driver

MAX7219 is an 8-digit multiplexed LED driver integrated circuit that has SPI interface. This circuit can be easily interfaced with any microcontroller. A single MAX7219 circuit can control up to 8 7-segment displayed in multiplexed scheme.

MAX7219 has several 8-bit registers that are to be programmed for displaying numbers on an LED display. For details, refer to the datasheet of MAX7219 available on:

<https://www.analog.com/media/en/technical-documentation/data-sheets/MAX7219-MAX7221.pdf>

### **Safety Instructions:**

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when circuit has been fully constructed.

**Program 1:**

```

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define CLK 18//CLK line of MAX7219
#define LOAD 19      //LOAD line of MAX7219
#define D_IN 21      //DIN line of MAX7219

#define SCAN_DIGITS 4    //number of digits to scan
#define INTENS_VAL 0x0F  //intensity of display. 0x04 for normal view
#define DASH 0x01       //code for '-' sign

//addresses of MAX7219 registers
#define DIG0 0x01
#define DIG1 0x02
#define DIG2 0x03
#define DIG3 0x04
#define DIG4 0x05
#define DIG5 0x06
#define DIG6 0x07
#define DIG7 0x08
#define DECODE 0x09
#define INTENS 0x0A
#define SCAN_LIMIT 0x0B
#define SHUT_DOWN 0x0C
#define DISPLAY_TEST 0x0F

typedef unsigned char UCHAR;
typedef unsigned int UINT;
typedef unsigned short USHORT;

// Delay for milliseconds
void delay_ms(UINT ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

```

```
/******
```

7-segment data buffer with all HEX characters from 0 to F.

```
bit no.      7  6  5  4  3  2  1  0
```

```
seg. no.    dp  a  b  c  d  e  f  g
```

```
*****/
```

```
char codes[16] = {0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70
                  ,0x7F,0x7B,0x77,0x1F,0x4E,0x3D,0x4F,0x47};
```

```
//function to transfer 16-bit packet to MAX7219 with MSB-first
```

```
void MAX7219_packet_write(char reg, char val){
```

```
    UINT packet;
```

```
    char i;
```

```
    gpio_set_level(LOAD, 0);    //LOAD line is low
```

```
    packet = (((unsigned int)reg)<<8) | (unsigned int)val);
```

```
    for(i=0; i<16; i++){
```

```
        if(packet&(0x8000>>i)){
```

```
            gpio_set_level(D_IN, 1);
```

```
            gpio_set_level(CLK, 1);
```

```
            gpio_set_level(CLK, 0);
```

```
        } //end if
```

```
        else{
```

```
            gpio_set_level(D_IN, 0);
```

```
            gpio_set_level(CLK, 1);
```

```
            gpio_set_level(CLK, 0);
```

```
        } //end else
```

```
    } //end for
```

```
    //LOAD line is high to latch data
```

```
    gpio_set_level(LOAD, 1);
```

```
    gpio_set_level(LOAD, 0);
```

```
} //end MAX7219_packet_write
```

```

//function configures MAX7219 for normal operation
void MAX7219_Config(){
    gpio_set_direction(D_IN, GPIO_MODE_OUTPUT);
        gpio_set_direction(LOAD, GPIO_MODE_OUTPUT);
        gpio_set_direction(CLK, GPIO_MODE_OUTPUT);

    gpio_set_level(CLK, 0);
    gpio_set_level(LOAD, 0);

    //set normal operation
    MAX7219_packet_write(SHUT_DOWN,1);

    //set normal operation
    MAX7219_packet_write(DISPLAY_TEST,0);

    //set no decode
    MAX7219_packet_write(DECODE,0x00);

    //set intensity value
    MAX7219_packet_write(INTENS,INTENS_VAL);

    //set number of digits scanned
    MAX7219_packet_write(SCAN_LIMIT,SCAN_DIGITS-1);
} //end MAX7219_Config

//function to send number to display
void NumOut(USHORT num){
    USHORT temp, d0, d1, d2, d3;

    temp = num;

    d0 = temp%10; //extract unit digit
    temp = temp/10;
    d1 = temp%10; //extract 10th digit
    temp = temp/10;

```



```

    d2 = temp%10; //extract 100th digit
    d3 = temp/10; //extract 1000th digit

    MAX7219_packet_write(DIG0, codes[d0]);
    MAX7219_packet_write(DIG1, codes[d1]);
    MAX7219_packet_write(DIG2, codes[d2]);
    MAX7219_packet_write(DIG3, codes[d3]);
} //end NumOut

/*****
void app_main() {
    UINT NUM = 0;
    MAX7219_Config();
    MAX7219_packet_write(DISPLAY_TEST, 1);
    delay_ms(1000);
    MAX7219_Config();
    for(;;){
        NumOut(NUM++);
        delay_ms(50);
    }
}
*****/

```

### In-Lab Exercise 1:

Make changes to Program 1 such that it displays a 4-digit random number (ranging from 0000 to 9999) after interval of one second.

Construct the circuit on breadboard and show the output to the lab instructor. Follow the safety instructions while demonstrating the task.

### In-Lab Exercise 2:

Make changes in Program 1 such that it displays counting from 0 to 999 on digits 0 to 2 (first 3 digits on right side) and from 999 to 0 on digits 5 to 7 (last 3 digits on left side). The two digits in the middle (digit 3 and digit 4) should remain off. Count should be updated after interval of 100 ms.

Construct the circuit on breadboard and show the output to the lab instructor. Follow the safety instructions while demonstrating the task.

## Matrix Keypad

A matrix keypad is an input device that can be used to input data into a processing device such as an MCU. A matrix keypad is arranged in the form of a pushbutton grid that are organized in rows and columns. Each pushbutton behaves as a simple pushbutton that can be configured as either pulled up or pulled down. In order to scan a keypad to identify any key pressed, all the columns are pulled up and rows are scanned one by one by providing logic 0 on one row at a time. The other condition is also possible when all the rows are pulled up and columns are scanned one by one by providing logic 0 on one column at a time.

A 4x3 (4 rows and 3 columns) matrix keypad is shown in Figure 6.3.



Figure 6.3

The keypad connector has 9 lines. 7 of them are for rows and columns whereas 2 are not used. The rows (R0, R1, R2, R3) and columns (C0, C1, C2) are numbered as follows.

Facing keypad up, the connector is numbered from right to left, starting from 1 to 9. Line 8 and 9 are not used. Remaining 7 lines are:

1      2      3      4      5      6      7

R1    R2    C2    R3    C0    R0    C1

## Program 2:

This program interfaces a 4x3 matrix keypad with ESP32 MCU board. The key input by the keypad is displayed on an I2C LCD.

Make 'include.c' file that contains the following text. This file contains the dependencies needed for interfacing keypad and PCF8574 I2C LCD.

Copy this file in the designated folder (refer to Lab 5).

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "PCF8574_LCD.c"
#include "KP4x3_ESP32.c"
```

Place the PCF8574\_LCD.c file designated folder (refer to Lab 5).

Make 'KP4x3\_ESP32.c' file that contains the following code. This file defines the functions and variables required for keypad scanning and key reading.

```
/******
```

This file defines the functions that scan a 4x3 matrix keypad and return the ASCII of that key.

CPU:                    ESP32  
Written By:            Engr. Usman Rafique  
Dated:                 Nov. 05, 2024

```
*****/
```

```
// Define GPIO pins for rows and columns
```

```
#define R0 GPIO_NUM_13
#define R1 GPIO_NUM_12
#define R2 GPIO_NUM_14
#define R3 GPIO_NUM_27
#define C0 GPIO_NUM_26
#define C1 GPIO_NUM_25
#define C2 GPIO_NUM_33
```

```
#define KP_Delay 10            //keypad key press delay in milliseconds
```

```
// 4x3 Keypad layout
```

```

char keys[4][3] = {
    {'1', '2', '3'},
    {'4', '5', '6'},
    {'7', '8', '9'},
    {'*', '0', '#'}
};

// Initialize GPIOs
void Keypad_Config() {
    // Set rows as output and columns as input
    esp_rom_gpio_pad_select_gpio(R0);
    esp_rom_gpio_pad_select_gpio(R1);
    esp_rom_gpio_pad_select_gpio(R2);
    esp_rom_gpio_pad_select_gpio(R3);
    esp_rom_gpio_pad_select_gpio(C0);
    esp_rom_gpio_pad_select_gpio(C1);
    esp_rom_gpio_pad_select_gpio(C2);

    gpio_set_direction(R0, GPIO_MODE_OUTPUT);
    gpio_set_direction(R1, GPIO_MODE_OUTPUT);
    gpio_set_direction(R2, GPIO_MODE_OUTPUT);
    gpio_set_direction(R3, GPIO_MODE_OUTPUT);
    // Default high
    gpio_set_level(R0, 1);
    gpio_set_level(R1, 1);
    gpio_set_level(R2, 1);
    gpio_set_level(R3, 1);
    gpio_set_direction(C0, GPIO_MODE_INPUT);
    gpio_set_direction(C1, GPIO_MODE_INPUT);
    gpio_set_direction(C2, GPIO_MODE_INPUT);
    // Enable pull-up resistors
    gpio_pullup_en(C0);
    gpio_pullup_en(C1);
    gpio_pullup_en(C2);
}

```

```

// Function to scan the keypad and return the key

```

```

char keypad_get_key() {
    char c0, c1, c2;

    //scan row0 only
        gpio_set_level(R0, 0);
    gpio_set_level(R1, 1);
    gpio_set_level(R2, 1);
        gpio_set_level(R3, 1);

    c0 = gpio_get_level(C0);
    c1 = gpio_get_level(C1);
    c2 = gpio_get_level(C2);

    if(c0==0 && c1==1 && c2==1)
        return(keys[0][0]);
    else if(c0==1 && c1==0 && c2==1)
        return(keys[0][1]);
    else if(c0==1 && c1==1 && c2==0)
        return(keys[0][2]);

    //scan row1 only
        gpio_set_level(R0, 1);
    gpio_set_level(R1, 0);
    gpio_set_level(R2, 1);
        gpio_set_level(R3, 1);

    c0 = gpio_get_level(C0);
    c1 = gpio_get_level(C1);
    c2 = gpio_get_level(C2);

    if(c0==0 && c1==1 && c2==1)
        return(keys[1][0]);
    else if(c0==1 && c1==0 && c2==1)
        return(keys[1][1]);
    else if(c0==1 && c1==1 && c2==0)
        return(keys[1][2]);

```

```

//scan row2 only
gpio_set_level(R0, 1);
gpio_set_level(R1, 1);
gpio_set_level(R2, 0);
gpio_set_level(R3, 1);

c0 = gpio_get_level(C0);
c1 = gpio_get_level(C1);
c2 = gpio_get_level(C2);

if(c0==0 && c1==1 && c2==1)
    return(keys[2][0]);
else if(c0==1 && c1==0 && c2==1)
    return(keys[2][1]);
else if(c0==1 && c1==1 && c2==0)
    return(keys[2][2]);

//scan row3 only
gpio_set_level(R0, 1);
gpio_set_level(R1, 1);
gpio_set_level(R2, 1);
gpio_set_level(R3, 0);

c0 = gpio_get_level(C0);
c1 = gpio_get_level(C1);
c2 = gpio_get_level(C2);

if(c0==0 && c1==1 && c2==1)
    return(keys[3][0]);
else if(c0==1 && c1==0 && c2==1)
    return(keys[3][1]);
else if(c0==1 && c1==1 && c2==0)
    return(keys[3][2]);

//this must be the very last statement

```

```

        else

            return '\0'; // Return null if no key is pressed
    }

// Task to display key presses
char Keypad_Read() {

    char key = '\0';

    while (key == '\0') {
        key = keypad_get_key();
    }
    vTaskDelay(pdMS_TO_TICKS(KP_Delay));

    return key;
}

```

Copy the following code in main.c. Run the code and show the output to the lab instructor.

```

#include "include.c"

#define LED_GPIO_NUM_2

// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

/*****/

void app_main() {

    char buff[20];
    char num;

    gpio_set_direction(LED, GPIO_MODE_OUTPUT);
    for(int j=0; j<6; j++){
        gpio_set_level(LED, 1);
        delay_ms(50);
        gpio_set_level(LED, 0);
        delay_ms(50);
    }
}

```

```

PCF8574_LCD_init();
Keypad_Config();

cursor_position(0,5);
lcd_print("Start");
cursor_position(1,5);
lcd_print("ESP32");

delay_ms(1000);
lcd_command(0x01);          // clear display
cursor_position(0,1);
lcd_print("Key pressed:");

for(;;){
    num = Keypad_Read();
    sprintf(buff,"%c ", num);
    cursor_position(1,7);
    lcd_print(buff);
}
}

```

### Lab Task 1:

Construct an embedded system (ES) using ESP32 that reads a temperature threshold value from keypad and controls an LED. A matrix keypad is used to read numbers. The required ES also has an RTC (DS1307) and an LM35 centigrade temperature sensor for measuring ambient temperature. Time and ambient temperature are to be displayed on a MAX7219 LED display stick. LED display should show the time in hours, minutes and seconds, separated with '-' sign. There should be a pushbutton that, when pressed, shows the current temperature on display.

When temperature exceeds the threshold, on-board LED turn on until it falls below the threshold. Temperature threshold is to be read from the keypad when ES is powered up.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.



### Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 7

### To Manipulate the External Interrupts and Timers on ESP32 using C++

#### Objectives

- Understanding external interrupt request (IRQ) mechanism
- Programming ESP32 for handling external IRQs and managing interrupt service routines (ISRs)
- Programming on-chip timers of ESP32 for generating delays and pulse width modulation (PWM) waveforms
- Interfacing DC motor for speed and direction control
- Interfacing servo motor

#### Pre-Lab Exercise

Read the details given below in order to comprehend the basic mechanism of interrupts on ESP32 and managing ISRs. In addition, using ESP32 timers for generating waveforms such as PWM. Then, generate the waveforms to control a servo motor with ESP32.

#### Interrupts

An interrupt is a mechanism in a digital circuit that causes a predefined change in the state of the circuit. In the context of a microprocessor or a microcontroller, an interrupt is a special input, either externally or internally, that brings the CPU in a new state. Upon acknowledging an interrupt, CPU performs following actions, generally in the order described below.

- ✓ It completes the instruction currently in execution.
- ✓ Pushes the address of next instruction to be executed on to the stack.
- ✓ Loads program counter with a specific address called ‘interrupt vector (IV)’.
- ✓ Starts executing program from the instruction pointed to by the IV until ‘return from interrupt (RFI)’ mechanism is not encountered.
- ✓ After encountering RFI, pops the address of instruction that is postponed from the stack in the program counter.
- ✓ Starts executing the program from this popped off address.

The program section that starts from the IV and ends at the RFI is called ‘interrupt service routine (ISR)’. Hence, an ISR is a special section in the program that is executed upon acknowledging the relevant interrupt. There can be multiple interrupts in a microprocessor or microcontroller, each

having its own IV. Interrupts can be triggered externally by changing logic level on the designated inputs. Or they can be triggered internally such as upon completion of reception of a byte by UART, completion of transmission of a byte by UART, overflow of a timer or other similar events.

Interrupts are among the most essential features of a microprocessor or a microcontroller that gives embedded system design engineer the liberty to handle untimely or asynchronous events.

### ➤ **Interrupts on ESP32 microcontroller**

ESP32 is a dual core 32-bit MCU, and each core has 32 interrupt sources. All GPIOs on ESP32 can be configured to trigger external interrupts. Each GPIO, if configured as external interrupt source, can trigger interrupt on a specific logic level or edge transition. If interrupt is triggered on a specific logic level, then it is called ‘level-triggered’. If interrupt is triggered on a specific edge, then it is called ‘edge-triggered’.

### ***Safety Instructions:***

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

### **Program 1:**

This program configures GPIOs on ESP32 to trigger external interrupts.

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

// Define GPIO pin numbers
#define INTERRUPT_PIN GPIO_NUM_34    // GPIO pin for external interrupt
#define LED_PIN GPIO_NUM_2           // On-board LED GPIO pin

// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

// Delay for microseconds
```

```

void delay_us(unsigned int i){
    esp_rom_delay_us(i);
}

// Interrupt service routine (ISR)
static void IRAM_ATTR GPIO_INT_ISR(void *arg) {
    gpio_set_level(LED_PIN, 1); // Turn on LED
    delay_us(500000);
    gpio_set_level(LED_PIN, 0); // Turn off LED
}

/*****/
void app_main() {
    // Configure LED pin as output
    esp_rom_gpio_pad_select_gpio(LED_PIN);
    gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);
    gpio_set_level(LED_PIN, 1); // Turn on LED
    delay_ms(200);
    gpio_set_level(LED_PIN, 0); // Turn off LED

    // Configure interrupt pin as input
    esp_rom_gpio_pad_select_gpio(INTERRUPT_PIN);
    gpio_set_direction(INTERRUPT_PIN, GPIO_MODE_INPUT);

    // Trigger on falling edge
    gpio_set_intr_type(INTERRUPT_PIN, GPIO_INTR_NEGEDGE);

    // Install GPIO ISR service
    gpio_install_isr_service(0);
    gpio_isr_handler_add(INTERRUPT_PIN, GPIO_INT_ISR, NULL);

    for(;;);
}

```

### In-Lab Exercise 1:

Make changes in Program 1 such that the two GPIOs (18 and 19) trigger two separate interrupts. GPIO18 should trigger a falling-edge-sensitive interrupt whereas GPIO19 should trigger a rising-

edge-interrupt. Falling-edge interrupt should turn on the on-board LED for 200 ms whereas rising-edge interrupt should turn on the on-board LED for 500 ms.

Use a breadboard to implement the circuit. Show your work to the instructor and record your observations in the box below.

### ➤ **Timers on ESP32 microcontroller**

ESP32 MCU has four 64-bit on-chip general-purpose timers embedded in the chip. Each timer has 16-bit prescaler and 64-bit auto-reload-capable up/down count capability. ESP32 chip contains two hardware timer groups, and each group has two general-purpose hardware timers.

Default clock for the timers is system's clock running at 280 MHz.

#### **Program 2:**

This program configures an on-chip timer on ESP32 to generate PWM waveforms.

```
#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "driver/ledc.h"

#define LEDC_TIMER LEDC_TIMER_0

#define LEDC_MODE LEDC_HIGH_SPEED_MODE

#define LEDC_OUTPUT_IO 18 // Set PWM output GPIO

#define LEDC_CHANNEL LEDC_CHANNEL_0

#define LEDC_FREQUENCY 1000 // Frequency in Hz

// Delay for milliseconds
```

```

void delay_ms(unsigned int ms){

    vTaskDelay(pdMS_TO_TICKS(ms));

}

// Delay for microseconds

void delay_us(unsigned int i){

    esp_rom_delay_us(i);

}

//configure LEDC timer and its channel

void LEDC_Timer_Config(){

// Configure the timer for the LEDC

    ledc_timer_config_t ledc_timer = {

        .speed_mode = LEDC_MODE,

        .timer_num = LEDC_TIMER,

        // Set duty resolution to 10 bits

        .duty_resolution = LEDC_TIMER_10_BIT,

        .freq_hz = LEDC_FREQUENCY,

        .clk_cfg = LEDC_AUTO_CLK

    };

    ledc_timer_config(&ledc_timer);

// Configure the LEDC channel

    ledc_channel_config_t ledc_channel = {

        .speed_mode = LEDC_MODE,

        .channel = LEDC_CHANNEL,

        .timer_sel = LEDC_TIMER,

        .intr_type = LEDC_INTR_DISABLE,

        .gpio_num = LEDC_OUTPUT_IO,

```

```

        .duty = 512, // 50% duty cycle (1024 max for 10-bit resolution)

        .hpoint = 0

    };

    ledc_channel_config(&ledc_channel);

}

//function sets dutycycle in percentage
void Set_DutyCycle(unsigned char dc_percent){

    ledc_set_duty(LED0_CHANNEL, LED0_CHANNEL, dc_percent*10.24);

    ledc_update_duty(LED0_CHANNEL, LED0_CHANNEL);

}

/*****/

void app_main(){

    LEDC_Timer_Config();

    unsigned char DC;

    while(1){

        for (DC = 0; DC <= 100; DC += 5) {

            Set_DutyCycle(DC);

            delay_ms(50);

        }

    }

}

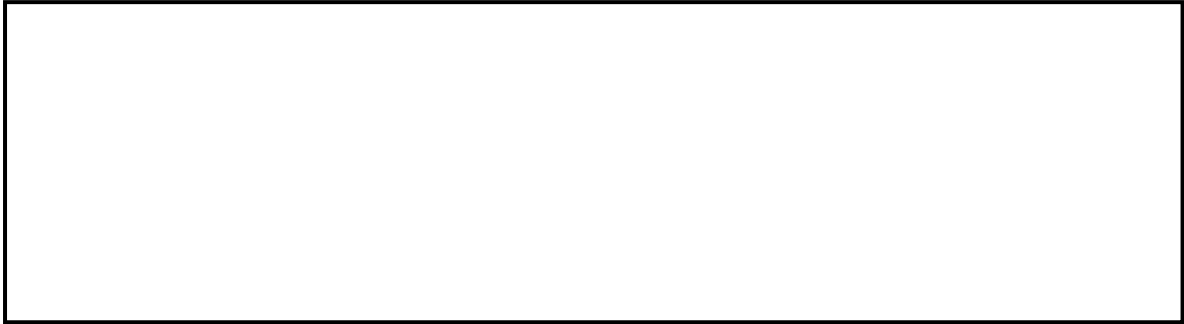
```

### In-Lab Exercise 2:

Make changes to Program 2 such that the GPIO23 is used as the PWM output. Set frequency of PWM equal to 5 kHz. Use an external LED connected with GPIO23 to show the response of PWM waveform.

Use a breadboard to implement the circuit. Show your work to the instructor and record your observations in the box below.



**Lab Task 1:**

Construct an embedded system using ESP32 that drives a DC motor. DC motor should run in both directions, and its speed is to be controlled through PWM. Use GPIO18 as input to select the direction of rotation of the motor. If GPIO18 is at logic 1, the motor should run clockwise, and if GPIO18 is at logic 0, the motor should run counterclockwise. The ES must contain a potentiometer that controls the dutycycle of PWM.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

**Lab Task 2:**

Construct an embedded system using ESP32 that drives a servo motor. Generate servo motor control waveform using on-chip timer.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 8

### To Construct C++ Code for on-chip Digital to Analog Converter of ESP32

#### Objectives

- Understanding digital to analog (DAC) converter operation
- Programming on-chip DAC of ESP32 to generate different periodic waveforms
- Programming on-chip DAC of ESP32 to generate random signals

#### Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of ESP32 on-chip DAC and programming it for generating analogue outputs.

#### Digital to Analog Converter (DAC)

A DAC is a hybrid circuit that converts digital numbers to analog signals. Digital numbers that are input to the DAC are converted to analog signals that are scaled within a pre-defined voltage range. The range of digital numbers is dependent on the number of bits for the DAC input. For example, a DAC with an 8-bit input range and 0V to 5V output voltage range can convert the numbers from 0 to 255 to produce analog signals from 0V to 5V. If the input number is 0 then output will be 0V and if the input number is 255 then output will be 5V. If this relationship is linear, then this DAC will be a 'linear DAC'. Figure 8.1 shows input-output relationship of a 4-bit linear DAC with input voltage range of 0V to 3.3V.

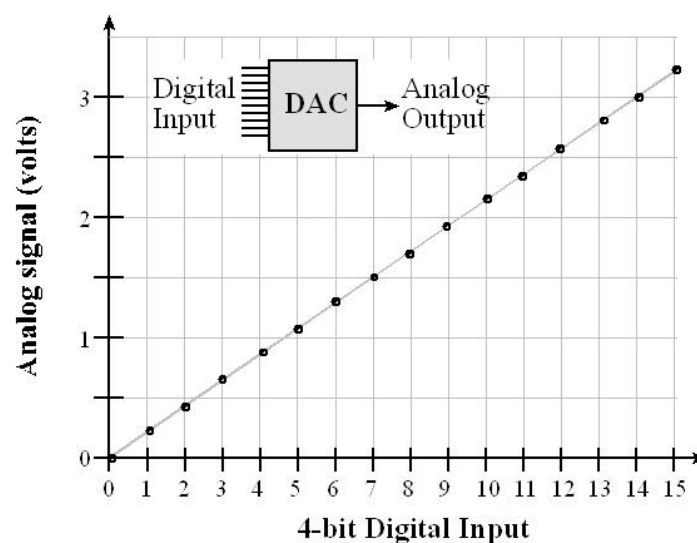


Figure 8.1

### ➤ DAC on ESP32 microcontroller

ESP32 has two 8-bit on-chip linear DAC channels, and their outputs are connected to GPIO25 (Channel 1) and GPIO26 (Channel 2). Each DAC can be configured for several output voltage ranges. The default output range is 3.3V.

#### ***Safety Instructions:***

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

#### **Program 1:**

This program configures on-chip DAC channel 1 to produce a ramp wave.

```
#include "freertos/FreeRTOS.h"
```

```
#include "freertos/task.h"
```

```
#include "driver/dac.h"
```

```
typedef unsigned char UCHAR;
```

```
typedef unsigned int UINT;
```

```
// Delay for milliseconds
```

```
void delay_ms(unsigned int ms){
```

```
    vTaskDelay(pdMS_TO_TICKS(ms));
```

```
}
```

```
// Delay for microseconds
```

```
void delay_us(unsigned int i){
```

```
    esp_rom_delay_us(i);
```

```
}
```

```
void app_main(){
```

```
    UCHAR i;
```

```
    dac_output_enable(DAC_CHANNEL_1); // GPIO 25 (DAC1)
```

```
    while(1){
```

```

        i = 0;
        while(i<255){
            dac_output_voltage(DAC_CHANNEL_1, i++);
            delay_us(1);           //delay for sampling period
        }           //end while
    }           //end while
}

```

### In-Lab Exercise 1:

Make changes in Program 1 such that a triangular waveform is generated at GPIO25.

Use a breadboard and a digital storage oscilloscope (DSO) to implement the circuit. Show your work to the instructor and record your observations in the box below.

### In-Lab Exercise 2:

Make changes in Program 1 such that a square wave of 1 kHz is generated at GPIO25.

Use a breadboard and a DSO to implement the circuit. Show your work to the instructor and record your observations in the box below.

### In-Lab Exercise 3:

Make changes in Program 1 such that a square wave of 5 kHz is generated at the GPIO25, and a triangular waveform of 1 kHz is generated at the GPIO26.

Use a breadboard and a DSO to implement the circuit. Show your work to the instructor and record your observations in the box below.

**Lab Task 1:**

Construct an embedded system using ESP32 that implements a function generator that produces a sine wave and a triangular wave. The sinewave is to be generated on GPIO25, and the triangular wave is to be generated on GPIO26. The frequency of the waveforms is to be controlled by a potentiometer connected with ADC. There must also be an LCD that displays the frequency of the output waveforms.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

**Lab Task 2:**

Construct an embedded system using ESP32 that generates a decaying exponential waveform on GPIO26. The output waveform has a voltage range from 0V to 3.3V.

Save your code and attach it to this Lab. Use breadboard to construct the circuit.

### Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 9

### To Construct C++ Code for Interfacing MicroSD Card with ESP32

#### Objectives

- To understand the data storage on microSD card
- To understand data write/read operation for a microSD card

#### Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of LCD and interfacing of LCD with microcontroller. It can be interfaced to perform different tasks in different combinations.

#### MicroSD Card

A microSD card is a non-volatile removeable storage medium that can be used to store data in files. The data from different sensors can be logged into files that can be read or retrieved further using PCs. A microSD card is formatted under either FAT-16 or FAT-32 file system to store data in files. The most common file that is used to store data is *.txt* file which is a binary file that accepts ASCII codes of the characters to be stored in it. A microSD card has an SPI interface and thus can be interfaced with a microcontroller such as ESP32 using SPI protocol.

#### *Safety Instructions:*

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

#### Program 1:

This program configures a microSD card for creating a *.txt* file on it, writing text in it, and then reading from it.

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <stdio.h>
#include <string.h>
#include "driver/gpio.h"
```



```

#include "esp_log.h"
#include "sdkconfig.h"
#define MOSI_PIN 23
#define MISO_PIN 19
#define SCK_PIN 18
#define CS_PIN 5

// Define SD card commands
#define CMD0 0x40    // GO_IDLE_STATE
#define CMD17 0x51   // READ_SINGLE_BLOCK
#define CMD24 0x58   // WRITE_SINGLE_BLOCK

static const char *TAG = "SD_SPI";

// Delay for milliseconds
void delay_ms(unsigned int ms){
    vTaskDelay(pdMS_TO_TICKS(ms));
}

// Bit-banged SPI send function
void spi_send(uint8_t data) {
    for (int i = 0; i < 8; i++) {
        gpio_set_level(MOSI_PIN, (data & 0x80) ? 1 : 0);
        gpio_set_level(SCK_PIN, 1);
        data <<= 1;
        gpio_set_level(SCK_PIN, 0);
    }
}

// Bit-banged SPI receive function
uint8_t spi_receive(void) {
    uint8_t data = 0;
    for (int i = 0; i < 8; i++) {
        gpio_set_level(SCK_PIN, 1);
        data <<= 1;
        if (gpio_get_level(MISO_PIN)) {

```

```

        data |= 0x01;
    }
    gpio_set_level(SCK_PIN, 0);
}
return data;
}

// Send command to SD card
void sd_send_command(uint8_t cmd, uint32_t arg, uint8_t crc) {
    gpio_set_level(CS_PIN, 0);
    spi_send(cmd);
    spi_send(arg >> 24);
    spi_send(arg >> 16);
    spi_send(arg >> 8);
    spi_send(arg);
    spi_send(crc);
    gpio_set_level(CS_PIN, 1);
}

// Initialize GPIO pins for bit-banged SPI
void spi_init() {
    gpio_set_direction(MOSI_PIN, GPIO_MODE_OUTPUT);
    gpio_set_direction(MISO_PIN, GPIO_MODE_INPUT);
    gpio_set_direction(SCK_PIN, GPIO_MODE_OUTPUT);
    gpio_set_direction(CS_PIN, GPIO_MODE_OUTPUT);

    gpio_set_level(CS_PIN, 1);
    gpio_set_level(SCK_PIN, 0);
}

// SD card initialization function
int sd_card_init() {
    spi_init();
    sd_send_command(CMD0, 0, 0x95); // CMD0: Reset SD card
    vTaskDelay(100 / portTICK_PERIOD_MS);

    // Check if SD card is in idle state

```

```

uint8_t response = spi_receive();
if (response != 0x01) {
    ESP_LOGE(TAG, "SD card not in idle state");
    return -1;
}
ESP_LOGI(TAG, "SD card initialized");
return 0;
}

// Write data to SD card (single block)
int sd_write_block(uint32_t block_addr, uint8_t *data) {
    sd_send_command(CMD24, block_addr, 0xFF);

    // Send start token (0xFE)
    spi_send(0xFE);

    // Send data block
    for (int i = 0; i < 512; i++) {
        spi_send(data[i]);
    }

    // Send dummy CRC
    spi_send(0xFF);
    spi_send(0xFF);

    // Check if the data was accepted
    uint8_t response = spi_receive();
    if ((response & 0x1F) != 0x05) {
        ESP_LOGE(TAG, "Data not accepted by SD card");
        return -1;
    }

    ESP_LOGI(TAG, "Block written successfully");
    return 0;
}

// Read data from SD card (single block)

```

```

int sd_read_block(uint32_t block_addr, uint8_t *buffer) {
    sd_send_command(CMD17, block_addr, 0xFF);

    // Wait for start token (0xFE)
    while (spi_receive() != 0xFE) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }

    // Read the data block
    for (int i = 0; i < 512; i++) {
        buffer[i] = spi_receive();
    }

    // Discard CRC
    spi_receive();
    spi_receive();

    ESP_LOGI(TAG, "Block read successfully");
    return 0;
}

// Write a file to the SD card
int sd_write_file(const char *filename, const uint8_t *data, size_t size) {
    // This is a simple block-level writing function.
    // Needed to integrate with a FAT32 file system.

    // Convert filename to a block address or find available space.

    if (sd_write_block(0, data) == 0) {
        ESP_LOGI(TAG, "File '%s' written successfully", filename);
        return 0;
    }

    ESP_LOGE(TAG, "Failed to write file '%s'", filename);
    return -1;
}

```

```

// Read a file from the SD card
int sd_read_file(const char *filename, uint8_t *buffer, size_t size) {
    // Similar to sd_write_file, this assumes block 0 for simplicity.

    if (sd_read_block(0, buffer) == 0) {
        ESP_LOGI(TAG, "File '%s' read successfully", filename);
        return 0;
    }

    ESP_LOGE(TAG, "Failed to read file '%s'", filename);
    return -1;
}

void app_main(void) {
    // Initialize SD card
    if (sd_card_init() != 0) {
        ESP_LOGE(TAG, "Failed to initialize SD card");
        return;
    }

    // Test write
    uint8_t write_data[512] = "Hello, SD card!";
    sd_write_file("test.txt", write_data, sizeof(write_data));

    // Test read
    uint8_t read_data[512] = {0};
    sd_read_file("test.txt", read_data, sizeof(read_data));

    // Print read data
    printf("Data read from SD card: %s\n", read_data);

    for(;;);
}

```

### Lab Task 1:

Create a C/C++ code that logs the data coming from on-chip ADC of ESP32 in a file named 'ADC.txt'. The file is to be stored on a microSD card interfaced with ESP32 using SPI link.

Save your code and attach it to this Lab.

### Lab Task 2:

Create a C/C++ code that logs the data coming from on-chip UART and on-chip ADC of ESP32 in a file named 'MyFile.txt'. The file is to be stored on a microSD card interfaced with ESP32 using SPI link. The data stored in the file should have the following layout.

ADC DATA	UART DATA
2056	128
56	247

Save your code and attach it to this Lab.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 10

### To Construct C++ Code for Interfacing Sensors and GPS Receiver with ESP32

#### Objectives

- To understand the communication protocol of ADXL345 triple-axis accelerometer
- To develop C/C++ code for receiving data from ADXL345
- To understand the National Marine Electronics Association (NMEA) messages from Neo-6m global positioning system (GPS) receiver
- To develop C/C++ code for receiving data from Neo-6m

#### Pre-Lab Exercise

Read the details given below to comprehend the basic operation of ADXL345 triple-axes accelerometer, NMEA sentences structure and interfacing the Neo-6m GPS receiver with ESP32 through UART interface.

#### ADXL345 Triple-axis Accelerometer

ADXL345 is a triple-axis accelerometer that measures acceleration of the device along x-axis, y-axis and z-axis. It measures both dynamic accelerations resulting from motion or shock and static acceleration, such as gravity, that allows the device to be used as a tilt sensor. Therefore, it can also be used as a gyroscope. The accelerometer can be communicated with through SPI or I2C. For simplicity of hardware, it is preferable to use I2C communication as it uses only two lines (SCK and SDA). The I2C address of ADXL345 is 0xA6 for writing and 0xA7 for reading. The data for three axes (x, y, and z) are stored in its 6 internal 8-bit registers. These registers are located at successive addresses starting from 0x32 to 0x37. Data related to each axis is a 16-bit value that is stored as 2 bytes in a pair of registers. Pinout of ADXL345 is shown in Figure 10.1.

The 6 registers and the data stored in them is as follows.

0x32 contains lower byte of x-axis

0x33 contains higher byte of x-axis

0x34 contains lower byte of y-axis

0x35 contains higher byte of y-axis

0x36 contains lower byte of z-axis

0x37 contains higher byte of z-axis

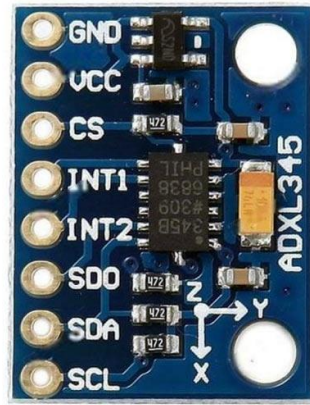


Figure 10.1

**Safety Instructions:**

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

**➤ Neo-6m GPS Receiver**

Global positioning system (GPS) is a US-owned global system of 24 satellites. These satellites are geo-stationary and are observed as static objects in the sky. They transmit positioning, navigation, and time (PNT) data towards the ground that can be received by GPS receivers like Neo-6m. The U.S. Space Force develops, maintains, and operates GPS.

Neo-6m is a simple GPS receiver that transmits NMEA sentences at regular interval of one second. Each NMEA sentence carries unique information related to PNT. The navigation data contains latitude and longitude coordinates of the geographical location where the GPS receiver is located. In addition, Universal Time Coordinated (UTC) is also transmitted. Navigation error of GPS data is less than 1 meter.

Neo-6m transmits data through UART that is configured at 9600 baud-rate as default value. The NMEA sentences received by the Neo-6m are shown below.

```
$GPRMC,110827.00,A,4107.32485,N,00831.79799,W,0.888,30.44,180724,,,A*4B
```

```
$GPVTG,30.44,T,,M,0.888,N,1.644,K,A*01
```

```
$GPGGA,110827.00,41XX.32485,N,00831.79799,W,1,07,0.99,123.1,M,50.1,M,,*48
```

```
$GPGSA,A,3,03,32,22,08,04,14,17,,,,,2.25,0.99,2.02*0A
```

```
$GPGSV,3,1,11,3,11,22,26,296,29,27,01,142,,32,17,042,23*48
```

```
$GPGLL,4107.32485,N,00831.79799,W,110827.00,A,A*7F
```



Each sentence starts with a \$ sign and ends with a line feed character (\n) whose ASCII code is 0x0A. Each sentence has a unique 5-character header i.e. GPGSV. Hence, this information can be used to parse a specific sentence from the data packet.

Pinout of Neo-6m is shown in Figure 10.2.

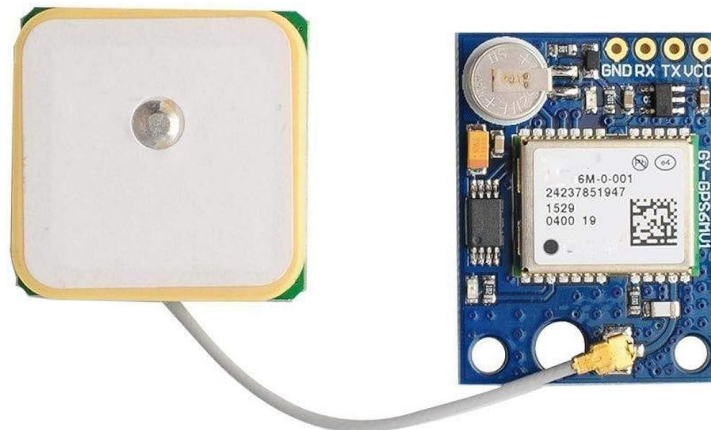


Figure 10.2

### Lab Task 1:

Create a C/C++ code that reads the readings from ADXL345 and shows on LCD. The data should be displayed according to the following format.

#### ADXL345 Accelerometer

**X: 23445**

**Y: 2556**

**Z: 3561**

Construct your circuit and show it to the lab instructor. Save your code and attach it to this Lab.

### Lab Task 2:

Create a C/C++ code that reads the NMEA sentence with header *GPRMC* from Neo-6m and displays it on LCD.

Construct your circuit and show it to the lab instructor. Save your code and attach it to this Lab.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 11

### To Construct C++ Code for Interfacing Stepper Motor with ESP32

#### Objectives

- To understand the working principle of 4-pole stepper motor
- To understand the hardware requirements of 4-pole stepper motor driver
- To develop C/C++ code for ESP32 to drive a 4-pole stepper motor for half-degree and full-degree rotation

#### Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of 4-pole stepper motor to drive it in full-step and half-step modes.

#### Stepper Motor

A stepper motor is a discrete motion mechanical device that can rotate precisely for a fixed number of degrees. A 4-pole stepper motor has two coils, internally fixed, to generate electromagnetic field. Each pole has two connectors for each end of coil. Therefore, there are 4 connectors for driving a motor by providing electric current to it. Two additional connectors are used for ground connections. A rotor is a magnet that aligns itself according to the coils energized at that moment.

Figure 11.1 shows the internal layout of a 4-pole stepper motor.

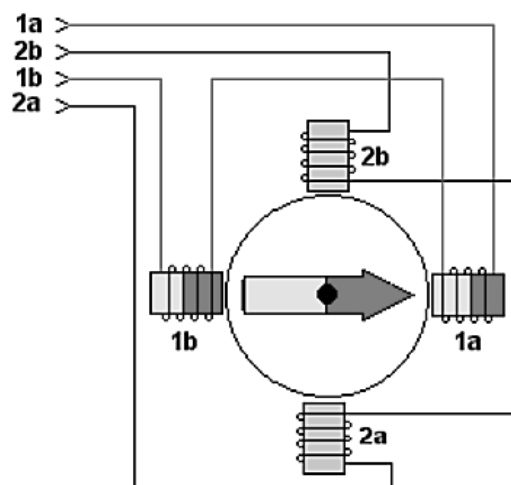


Figure 11.1

In order to drive a stepper motor, digital pulses are to be provided on its coil that supply the required current to magnetize the coil. A digital controller such as a microprocessor or a microcontroller can

provide these pulses. However, the GPIOs of these controllers cannot source enough current. Hence, a darlington-based driver circuit is needed. The schematics of the darlington-based driver is shown in Figure 11.2.

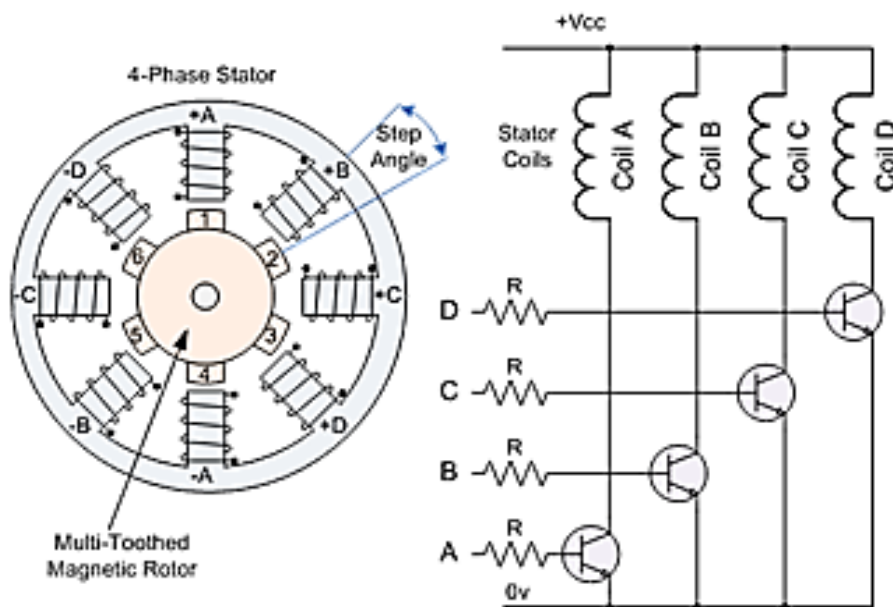


Figure 11.2

The digital pulses required to be generated by the ESP32 are shown in Figure 11.3. Hence, 4 GPIOs are needed to generate these pulses.

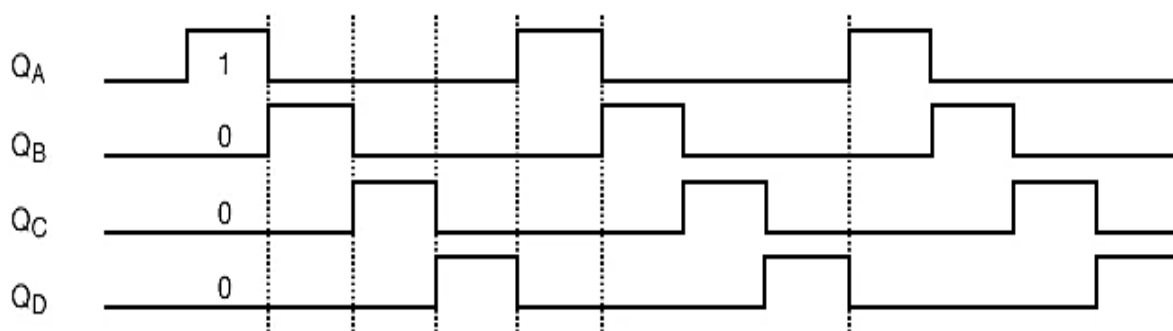


Figure 11.3

### Safety Instructions:

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing a circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

**Lab Task 1:**

Create a C/C++ code that drives a 4-pole stepper motor in full-step mode with variable interval between each step. The interval is to be controlled by a potentiometer that is interfaced with the on-chip ADC. The minimum interval should be 50ms and the maximum interval should be 2 seconds. Construct your circuit and show it to the lab instructor. Save your code and attach it to this Lab.

**Lab Task 2:**

Create a C/C++ code that drives a 4-pole stepper motor in half-step mode. The motor should run for a given number of degrees that is entered via a potentiometer interfaced with on-chip ADC. The number of degrees entered by the potentiometer should be in the range of 0.9 to 360 degrees. There must be a pushbutton that is used to enter the degrees from the potentiometer. In addition, the embedded system must also contain an LCD that shows information such as entered degrees and remaining degrees.

Construct your circuit and show it to the lab instructor. Save your code and attach it to this Lab.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## LAB # 12

### To Interface Radio Frequency (RF) Module with ESP32 using C Programming

#### Objectives:

1. Understanding the interfacing of radio frequency (RF) transceiver modules operating in ISM band with ESP32
2. Understanding AT commands
3. Developing C/C++ code for ESP32 to create a radio link between two ESP32 devices
4. Developing C/C++ code for ESP32 to create a radio link among multi-node RF network controlled by ESP32 devices

#### ➤ Industrial, Scientific and Medical (ISM) bands

A portion of electromagnetic spectrum used for common data communication among digital devices is called ISM band. This band of EM spectrum is free-of-license and can be freely used all over the world for scientific and engineering radio communication. Commonly used ISM bands are 433 MHz, 2.4 MHz and 5 MHz. Several devices, called RF transmitters and RF receivers, are available that exploit these bands to allow digital communication among several devices. If an RF communication device is capable of transmitting and receiving data simultaneously, then the device is termed as 'RF transceiver'.

These RF transceivers are the backbone of IoT-governed embedded systems and several other remote data communication applications. One such RF transceiver is HS12 that operates at 433 MHz and has UART interface for receiving and transmitting binary data. The transceiver is capable of transmitting data at maximum power of 100mW (20dBm). The default transmitting power is 0dBm or 1 mW that can cover the range of around 500 meters in line-to-line communication. The sensitivity of the device can be enhanced using a helical antenna. Figure 1 shows the HC12 module with antenna.

Figure 12.1 shows that the HC12 has two power lines (VCC and GND) that can sustain with 3.2V DC to 5.2V DC. The byte that is to be radio-transmitted needs to be loaded in the HC12 through RX line, following UART interface protocol. The byte received by the device can be read from TX following UART protocol. The default baudrate of HC12 is 9600 with 8N1 frame format. The logic level 0 at SET input configures the devices for responding to AT commands. The SET input needs to be at logic 1 for RF communication.

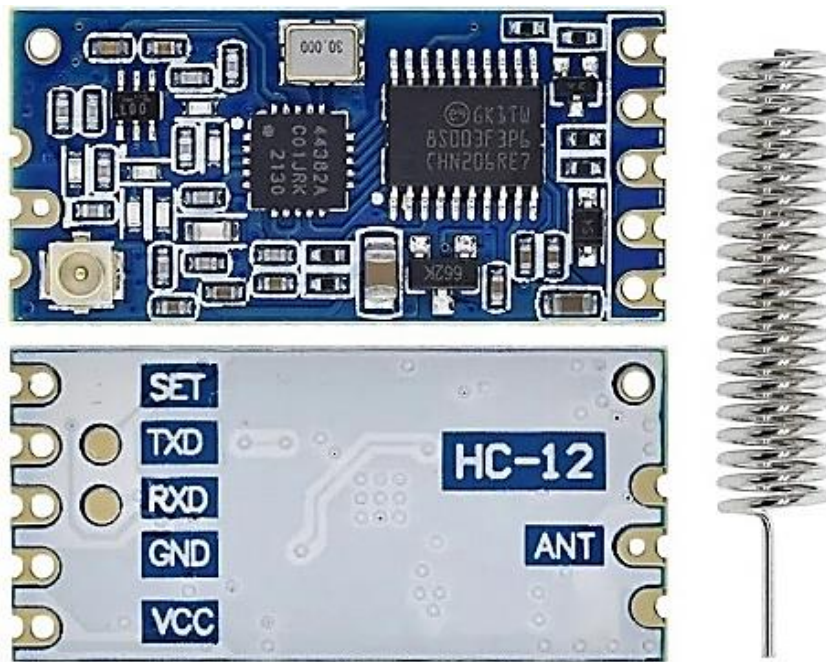


Figure 12.1

### ➤ AT Commands

AT commands, short for ‘ATtension’ commands, is a set of commands that are issued by a master to the slave for configuring the slave to work in a required mode of operation. The master issues the commands and slaves respond with sentences called ‘reply’. Both the master and the slave follow UART protocol for exchanging AT commands and replies.

Each AT command and the reply is a sequence of ASCII characters. A master, generally an MCU, initiates the communication with slave, the device that is to be configured. HC12 can work as a slave and be configured for required operation. HC12 can be configured to work at 100 different channels, each separated by the gap of 400 kHz. In order to communicate data between two HC12 devices, each of them must be configured with same channel number. However, the baudrates can be different.

HC12 can respond to the following AT commands. Each AT command must be terminated with ‘\n’ and ‘\r’ with ASCII codes of 0x0A and 0x0D, respectively. Similarly, each reply by the slave is terminated with ‘\n’ and ‘\r’.

1. **AT** This is a test command that replies OK by the slave.
2. **AT+BXXXX** This command sets the baudrate of the slave. The new baudrate will be effective after powering off the device and repowering it. For example, to configure the slave at baudrate 4800, the command will be AT+B4800. The slave replies with OK+B4800 after successful processing of command.



3. **AT+CXXX**      This command sets the channel number of the slave. The slave replies with **OK+CXXX**. For example, if slave is to be configured with channel no. 45, then command will be **AT+C045** and reply will be **OK+C045**.
4. **AT+SLEEP**      This command brings the HC2 in sleep mode. This mode can be invoked in device when no RF communication is in operation. The slave replies with **OK+SLEEP**. Sleep mode can save a considerable amount of power.

***Safety Instructions:***

- Disconnect the ESP32 board from the computer while setting up the circuit.
- Constructing circuit around ESP32 while it is connected with computer can damage both the board and the computer.
- Only connect the ESP32 board with the computer when the circuit has been fully constructed.

**Lab Task 1:**

Construct an embedded system that consists of ESP32-driven radio link. One ESP32 acts as transmitter and the other as receiver. Using default configurations of HC12, create a C/C++ code that transmits a string of upper-case alphabets from transmitter to slave. Upon receiving the string, slave sends back the same characters but in lowercase. An LCD is interfaced with the receiver that prints the received string.

Construct your circuit and show it to the lab instructor. Save your code and attach it to this Lab.

**Lab Task 2:**

Construct an embedded system that consists of an ESP32-driven wireless network. There must be a Master named **M**, that controls the network. There must be two slaves, **S01** and **S02**, each configured with a different channel. The Master has channel number 10.

Each slave is interfaced with a sensor that acquires data from them. The slaves send data to the master. The master interfaces with an LCD that shows the incoming data from each slave on LCD.

Construct your circuit and show it to the lab instructor. Save your code and attach it to this Lab.

## Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

**Instructor's Signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

## Hardware Lab Evaluation Rubric

Criteria	Exceeds Expectations (4)	Meets Expectation (3)	Developing (2)	Unsatisfactory (1)
<b>Setup of experiment and implementation (hardware/simulation)</b>	Can identify new ways to set up and implement the experiment without assistance and with detailed understanding of each step	Can fully set up the experiment with successful implementation without assistance	Can setup most of the experiment with some implementation without assistance	Can't set up the experiment without assistance
<b>Follow the procedure/design process</b>	Follows the procedure/design process completely and able to simply or develop alternate procedure/design	Follow the procedure/design process completely	Follows most of the procedures/design process with some errors or omissions	Doesn't follow the procedure/design process
<b>Experimental results</b>	Able to achieve all the desired results with new ways to improve measurements/synthesis	Able to achieve all the desired results	Unable to achieve all the desired results in implementation	Unable to get the results
<b>Safety</b>	Extremely conscious about safety aspects	Observes good laboratory safety procedures	Unsafe lab procedures observed infrequently	Practice unsafe, risky behaviors in lab
<b>Viva</b>	Able to explain design, simulation, implementation and fundamental concepts correctly and provide alternative solutions	Able to explain design, simulation, implementation and fundamental concepts correctly	Able to explain some design and relevant fundamental concepts	Unable to explain design and answer relevant fundamental Concepts

## Software Lab Evaluation Rubric

Criteria	Exceeds Expectations (4)	Meets Expectation (3)	Developing (2)	Unsatisfactory (1)
<b>Ability to use software</b>	Student was familiar with the software and was able to use additional features of the software that were not available in instruction set.	Student was familiar with the software and required minimal help from the instructor to perform the experiment	Student demonstrated an ability to use the software but required assistance from the instructor	Student demonstrated little or no ability to perform experiment and required unreasonable amount of assistance from instructor
<b>Ability to follow procedure and/or design a procedure for experiment</b>	<p>Student followed the instructions with no assistance</p> <p>student performed additional experiments or tests beyond those required in instructions</p> <p>if procedure to accomplish an objective in not provided, the student developed a systematic set of tests to accomplish objective</p>	<p>Student followed instructions in the procedure with little or no assistance</p> <p>if procedure was provided, the student was able to determine an appropriate set of experiments to run to produce usable data and satisfy the lab objectives</p>	<p>Student had difficulty with some of the instructions in the procedure and needed clarification from the instructor</p> <p>if procedure was not provided, the student needed some direction in deciding what set of experiments to perform to satisfy the lab objective</p>	<p>Student had difficulty reading the procedure and following directions</p> <p>if procedure was not provided, student was incapable of designing a set of experiments to satisfy given lab objective</p> <p>the data taken was essentially useless</p>
<b>Ability to troubleshoot software</b>	<p>Student developed a good systematic procedure for testing software code that allowed for quick identification of problems</p> <p>student good at analyzing the data</p>	Student demonstrated the ability to test software code in order to identify technical problems, and was able to solve any problems with little or no assistance	Student was able to identify the problems in software code but required some assistance in fixing some of the problems	Student demonstrated little or no ability to troubleshoot software code for the lab.
<b>Q &amp; A</b>	Able to explain program design and fundamental concepts correctly	able to explain most of the program design and relevant fundamental concepts	able to explain some program design and relevant fundamental concepts	unable to explain program design or answer relevant fundamental concepts

## Lab Report Evaluation Rubric

Criteria	Exceeds Expectations (4)	Meets Expectation (3)	Developing (2)	Unsatisfactory (1)
<b>Data Presentation</b>	Student demonstrates diligence in creating a set of visually appealing tables and/or graphs that effectively present the experimental data	Experimental data is presented in appropriate format with only a few minor errors or omissions	Experimental data is presented in appropriate format but some significant errors are still evident.  Tables could be better organized or some titles, labels or units of measure are missing.	Experimental data is poorly presented.  Graphs or tables are poorly constructed with several of the following errors: data is missing or incorrect, units are not included, axis not labeled, or titles missing.
<b>Data Analysis</b>	Student provides a very focused and accurate analysis of the data. All observations are started well and clearly supported by the data	Student has analyzed the data, observed trends, and compared experimental results with theoretical results  Any discrepancies are adequately addressed  All expected observations are made.	Student has analyzed the data, observed trends, and compared experimental results with theoretical results  Any discrepancies are not adequately addressed  Some observations that should have been made are missing or poorly supported	Student has simply restated what type of data was taken with no attempt to interrupt trends, explain discrepancies or evaluate the validity of the data in terms of relevant theory  Student lacks understanding of the importance of the results
<b>Writing Style</b>	Lab report has no grammatical and/or spelling errors.  All sections of the report are very well written	Lab report has very few grammatical and/or spelling errors  the sentence flow is smooth	Lab report has some grammatical and/or spelling errors and is fairly readable  Student makes effective use of technical terms	Lab report has several grammatical and/or spelling errors and sentence construction is poor